Table of Contents

ANALYSIS 3

1. IDENTIFICATION 3
   a. Problem Description 3
   b. Why the problem is suitable for a computational approach 3
2. STAKEHOLDERS 3
   a. Identification of stakeholders 3
   b. How the solution is appropriate to the stakeholder's needs 3
3. RESEARCH 4
   a. Interview with Dr Monty Lyman and Dr Rishika Sinha 4
   b. Survey about computerised melanoma detection 5
   c. Analysis of survey results 6
   d. Analysis of SkinVision 9
4. PROPOSED SOLUTION 13
   a. Choice of device 13
   b. Success Criteria 13
   c. Features 15
   d. Hardware Requirements 16
   e. Software Requirements 16
   f. General Limitations 17
   g. Software Limitations 17
   h. Hardware Limitations 17

DESIGN 19

1. DECOMPOSITION 19
   a. Problem Decomposition 19
   b. Structure Chart 21
   c. Algorithmic overview 22
2. SOLUTION MOCKUP 28
   a. Usability Measures 28
   b. Balsamiq Mockup 29
   c. Stakeholder feedback 33
3. APP DATA STRUCTURE 34
   a. UML class diagrams 34
   b. Validation 34
4. TESTING PLAN 35
   a. Testing during development 35
   b. Post development test plan 36

Analysis

1. Identification

a. Problem Description

Skin cancer is the most prevalent type of cancer today. Melanoma is the most common type of skin cancer, and accounts for around 75% of skin cancer deaths. There were around 325,000 new cases reported just in 2020, with around 60,000 deaths in 2020 directly attributed to melanoma [ref_1].

The problem is further propagated in less developed countries, where people do not have reliable access to good healthcare. Due to this, they are often not diagnosed and once they realise, they have a serious disease they are not either able to get to good enough healthcare, or more commonly are not able to financially afford to get treatment.

Melanoma is a deadly disease, but if it is recognised and diagnosed early most melanoma cases can be cured with minor surgery. The minor surgery is not only much cheaper (less of a financial burden), but is also less technically advanced (so the treatment can be done with less specialised equipment and doesn't require such an experienced doctor).

b. Why the problem is suitable for a computational approach

As melanoma is visible to the naked eye, a camera can be used to take a picture and then the product can run an algorithm to give a prediction about whether the picture contains melanoma or not, and with what certainty. Based off that prediction, the product would be able to provide some kind of further advice to the user, such as a direct reflection of advice given by trusted sources such as the NHS, and guidelines to self-diagnose to provide an additional layer of certainty.

This is particularly suited to a computational approach as it involves a lot of data crunching with image processing and machine learning (and subsequently a lot of iteration). With modern day processors this can be done relatively easily. As well as this, modern-day cameras can take high quality images of human skin, allowing for predictions to be made with as much data as needed – increasing the reliability and accuracy of the algorithm. A computational approach will also allow for classification of skin diseases very quickly, allowing for more diagnoses to be possible in a shorter amount of time.

2. Stakeholders

a. Identification of stakeholders

The stakeholders of my product fit into two main categories – those who will use the app and those who will be aided by the app's performance.

The first category includes the people who would use this app. These will be people who are concerned about their health, and want to check their skin for melanoma. They will use the app by taking pictures of the skin anomaly and uploading it into the app. Then the app will run an algorithm to make a prediction about the skin, whether is malignant or benign (active, cancerous or not active, not cancerous). The app will also show its certainty in its prediction, letting the user know if their case is especially serious. Furthermore, the app will give up to date advice from reliable organisations, so that the users know what the next steps for them would be given their condition.

The second category is the people who will be aided by the app's performance, this would mainly be healthcare organisations and doctors. With this app they could help diagnose melanoma over the phone, or even via email, which would save time, transportation, money and may allow the doctor to deal with more cases in a given amount of time. Doctors would also be able to use this app to monitor the state of a patient's skin without needing in-hospital check-ups, this would be especially helpful for repeat cancer patients, and for the cases the doctor is unsure about.

b. How the solution is appropriate to the stakeholder's needs

This product is appropriate to their needs as it provides a wider range of people with access to free diagnosis of melanoma. It will allow many more people to self-diagnose with a certain level of confidence without having to go to healthcare professionals, which not only takes a lot of time and is often not possible but also costs a lot in a lot of countries. As well as this the medical professionals will have the mundane and logical work of diagnosis cut out, and so they will be able to spend their time on treating people rather than diagnosis.

The app will generally make the diagnosis of melanoma much easier and quicker. Therefore, it will allow more people to self-diagnose with ease, and make it a process that is available reliably for all.

3. Research

a. Interview with Dr Monty Lyman and Dr Rishika Sinha

I decided to interview Dr Monty Lyman and Dr Rishika Sinha, who are very knowledgeable in the field of dermatology. I believe getting an insight into the field of dermatology and the current state of melanoma diagnosis would help me understand the problem better, and so create a better solution. As the interviews were quite long, I have written a summary of what they said, only mentioning the most important details.

"Initial triage and diagnosis of melanoma is done through visual inspection by health professionals who are not melanoma specialists. This results in many unnecessary referrals to an already overstretched dermatology service, as well as the more serious problem of missing skin cancer. Artificial Intelligence provides a huge opportunity in aiding clinical decisions surrounding the diagnosis of this deadly disease." – summarising quote from Dr Lyman, author of The Remarkable life of the Skin.

α. *Current state of melanoma diagnosis*

Dr Lyman –

> People have to go to doctors that are usually not specialists (ie GPs) for an initial inspection, and if the GP believes that there is sufficient evidence of melanoma, the patient is referred to a specialist. Unfortunately the waiting times for specialists inspection are quite long, and for melanoma detection time is of the essence. This waiting time is usually due to false positives from the GPs, and as well as this sometimes GPs miss positive cases.

Dr Sinha –

> Go for a checkup with a GP, and if the GP has sufficient evidence or the patient is unsatisfied the GP refers the patient to a specialist. Suspected melanoma should be acted on quickly as it can cause permanent damage quite early. The patient should go for a checkup within 2 weeks, and if positive get treatment within another 30 days.

β. *Would a product that helps aid diagnosis be beneficial?*

Dr Lyman –

> Yes, there have been studies done that show the detection of melanoma by eye often results in false positives and sometimes misses positive cases, even when the detection is carried out by GPs. The current guidance for self detection is using the 'ABCDE' rule, which is good for a general inspection but the method is just not consistent enough.

Dr Sinha –

> It is not reasonable to believe the product will be able to diagnose with 100% accuracy just by looking at the mole. The product would be beneficial as an aid to diagnosis, but it should not replace diagnosis by professionals. Another problem is false diagnosis via technology often leads to lots of patients being overly paranoid and not believing the specialists when they say that they do not have a problem. This ends up with unnecessary treatment costing time and money – something that could be spent actually saving someone with melanoma.
>
> As well as images the product should determine risk factors (specified by dermatological institutes, such as type of skin, immuno-status, evolution of mole and family history), this will allow for more consistently accurate predictions, and will be a better diagnosis.

γ. *Do today's smartphones have cameras that are high enough resolution to make inferences from images of possible melanoma?*

Dr Lyman –

I am not very aware of the capabilities of smartphones today, but for proper inspection of melanoma a dermascope is used. It magnifies the skin and illuminates it so that the different colours of the melanoma are easier to see.

Dr Sinha –

Modern day smartphone cameras (such as on the iPhone) are good enough, but a lot depends on how good the algorithm that is making the prediction is.

δ. *Are there any other widespread skin diseases that are detected in a similar way that pose a greater threat than melanoma?*

Dr Lyman –

There are three types of skin cancer that are relatively common. The most common is Basal Cell Carcinoma, it is usually benign and doesn't spread or have much of an effect on health. Therefore, it would be helpful to know of its existence, but it isn't needed. The second most common is Squamous Cell Carcinoma, this is also similar to BCC but can be slightly more dangerous as it spreads a bit more. The third most common is Melanoma. Melanoma is the most dangerous due to the fact it spreads very quickly. If not acted upon quickly melanoma can even prove to be fatal and occurs in people of all ages, so a product to help detect melanoma would have the greatest impact.

Dr Sinha –

Almost identical response to Dr Lyman's.

ε. *Would you recommend any organisations that provide data/information about melanoma to use whilst developing this product?*

Dr Lyman –

Yes, a paper by A.Esteva et al – "Dermatologist-level classification of skin cancer with deep neural networks.". I read it as part of my research whilst writing my book and it provides great insight about how to predict melanoma well from images. As well as this a study by J.Dinnes et al – "How accurate is visual inspection of skin lesions with the naked eye for diagnosis of melanoma in adults" [ref_2] – shows why there is a problem and technology can help solve it.

Dr Sinha –

The New Zealand dermatology department is arguably the best in the world and provides great information [ref_3]. It is often used for training doctors.

I asked about the reliability of ISIC [ref_4] (The International Skin Imaging Collaboration), and both Dr Lyman and Dr Sinha said that it is a reliable organisation and I should use their data for training should that be the need.

b. Survey about computerised melanoma detection

I decided to survey as many people as possible in different scenarios to learn about their current understanding about melanoma detection. The respondents will be the eventual users of the product, and so should give me a good idea of what the average user knows about melanoma, what they hope to gain from a product to help diagnose melanoma, and what would make them trust a product that diagnoses them.

I collected 297 responses for my survey, I believe this number of responses can allow me to confidently infer from the results. The questions I asked were as follows:

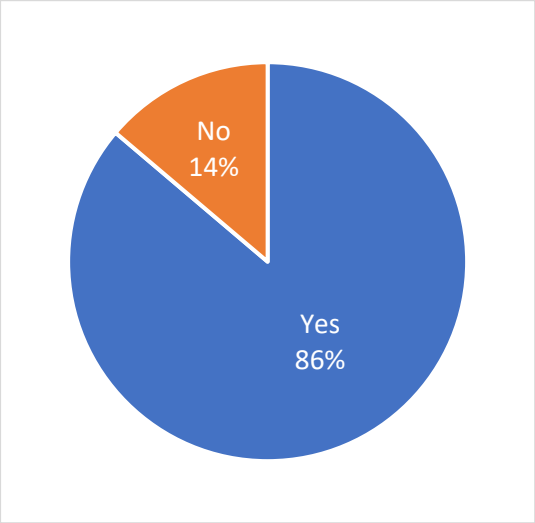
1. Before this survey, were you aware of the existence of melanoma, or any other type of skin cancer?
2. How often do you go for a general medical check-up (include any visits to a GP or doctor)?
3. During these check-ups are you screened for melanoma, or any type of skin cancer?

4. Do you know how you could self-diagnose melanoma? (Could you tell if someone has melanoma by looking at their skin?)
5. Arrange these features in order of importance to you if there was an app to help diagnose melanoma (highest is most important, lowest is least important).
6. On a scale of 1-10, how much would you trust a computer prediction?
7. What would make you trust a computer's prediction more?

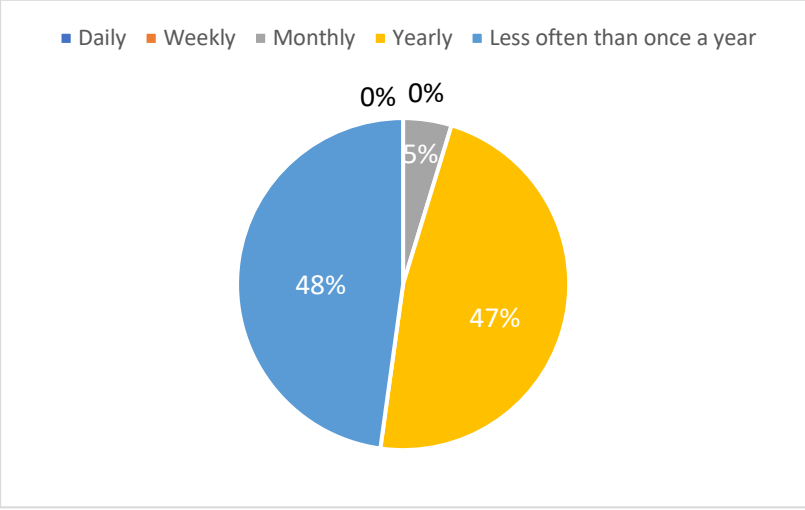
Results: https://forms.office.com/Pages/AnalysisPage.aspx?id=XN7vxi-BKEePctvBoUB1ACJojeaWe45Dq6cxiBimoOpUQlJTUzQ1SFQ3ODFTUURDWDE0TVRBRzZPVC4u&AnalyzerToken=jnzWAkBk7QwDEduVl9AOHxgvZrZPBMNH

c. Analysis of survey results

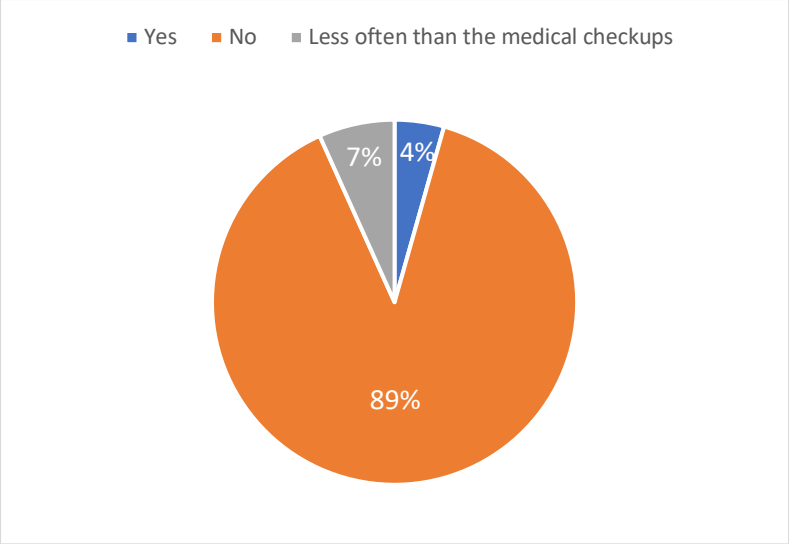
1. Before this survey, were you aware of the existence of melanoma, or any other type of skin cancer?



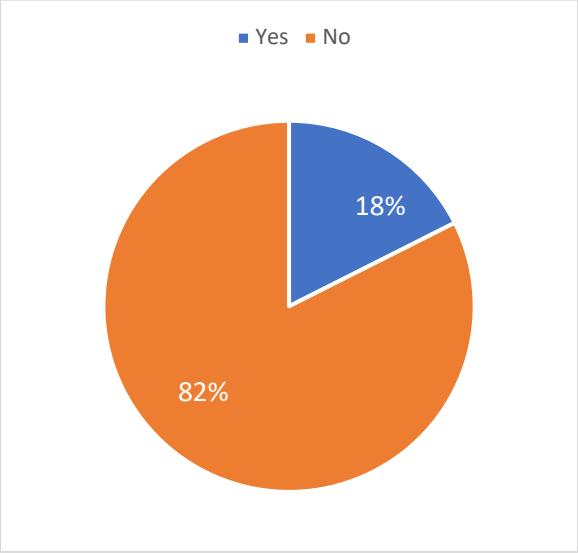
2. How often do you go for a general medical check-up (include any visits to a GP or doctor)?



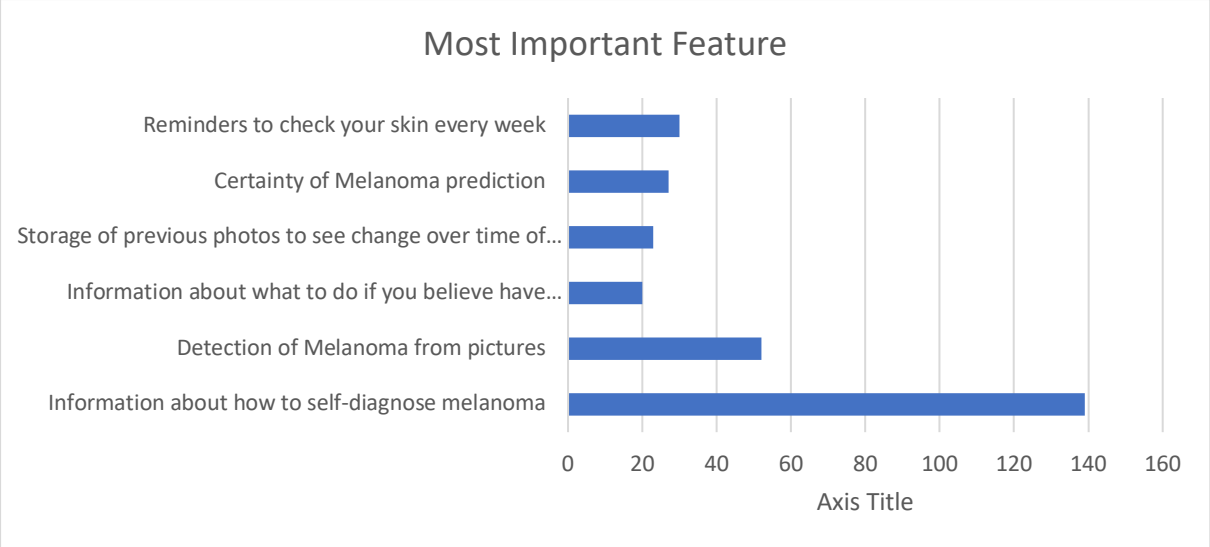
3. During these check-ups are you screened for melanoma, or any type of skin cancer?

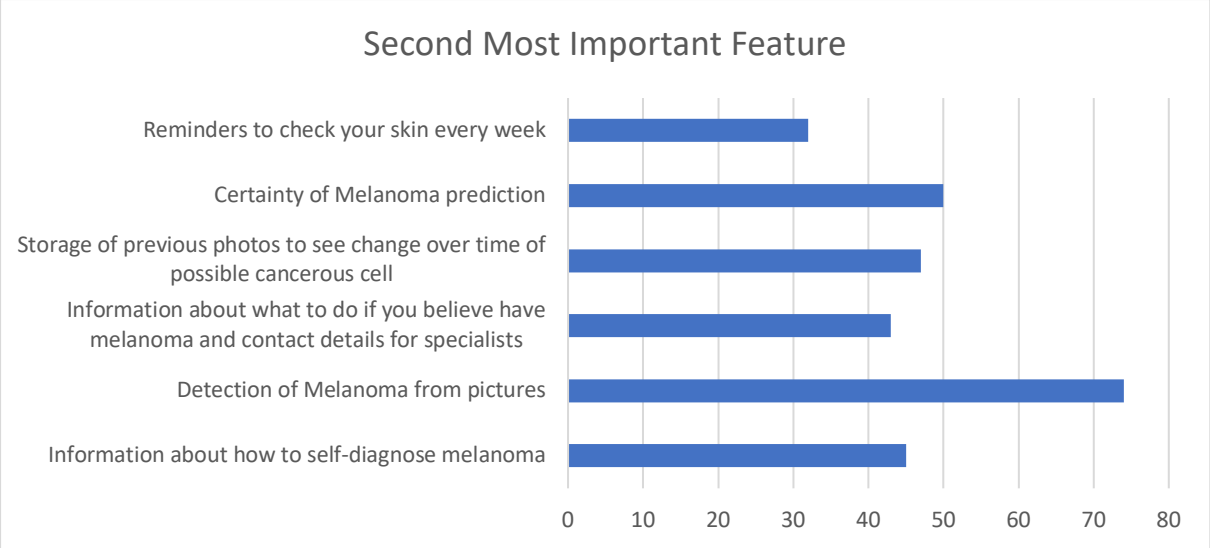


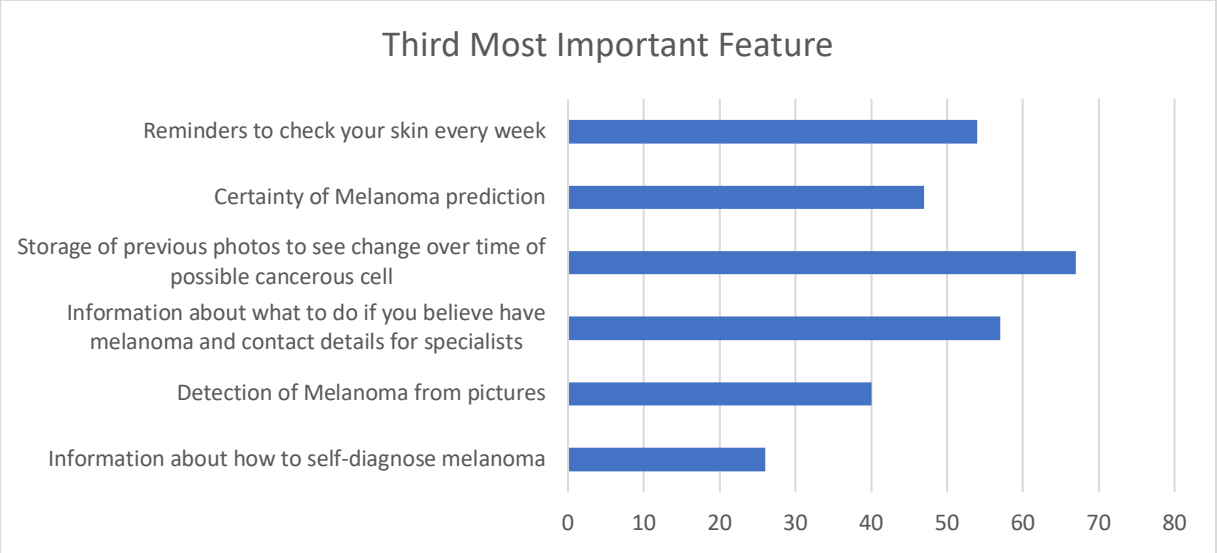
4. Do you know how you could self-diagnose melanoma? (Could you tell if someone has melanoma by looking at their skin?)



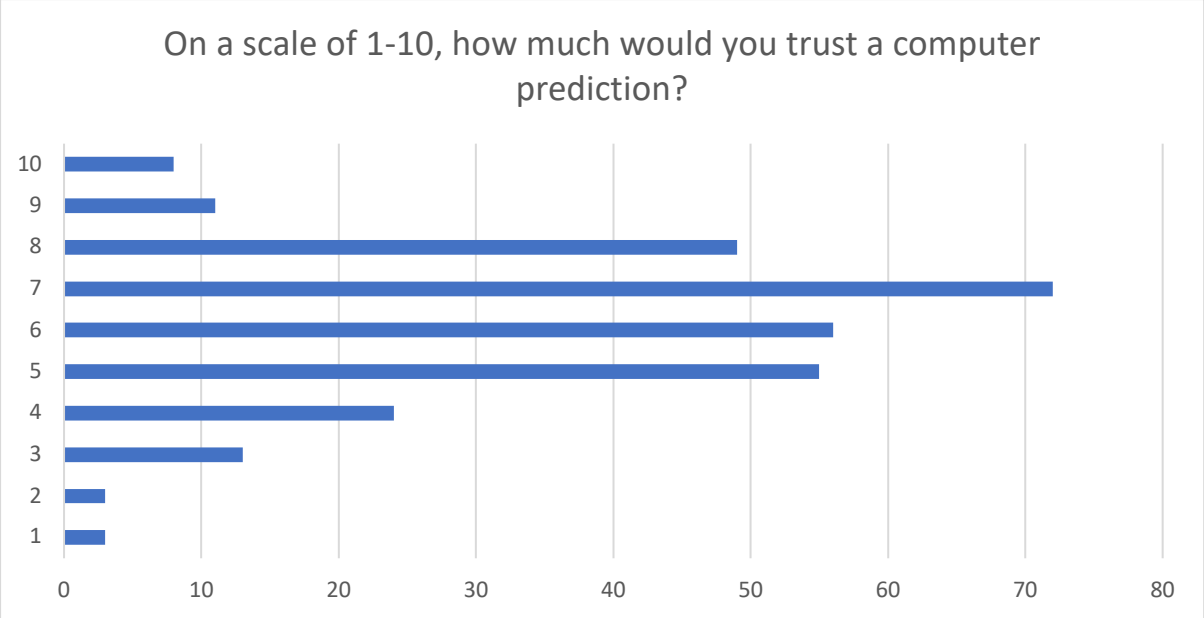
5. Arrange these features in order of importance to you if there was an app to help diagnose melanoma (highest is most important, lowest is least important).







6. On a scale of 1-10, how much would you trust a computer prediction?



7. What would make you trust a computer's prediction more?

Brief summary of most common themes –

- Explanation of algorithm that diagnoses the user, so that the user can understand what is diagnosing them. Helps build trust with the user.
- Further checking with doctors or specialists.
- Study showing the accuracy of algorithm, and possible success cases.
- Qualified doctors and specialists backing app.
- Ensuring no data is leaked or stored without consent.

Key Findings

Here I will go over the key findings and main takeaways for a solution to this problem. The justification will be in square brackets, referring to the survey result that justifies this.

α. Many people are aware of skin cancer and its problems [1], yet are not being screened for melanoma that often [3] despite having access to doctors at least once a year [2].

β. Many people are not able to self-diagnose melanoma [4]. Therefore, an app that can provide information about diagnosis and is also able to diagnose will be useful for the majority of people.

γ. The majority of respondents of the survey believed that information about how to self-diagnose melanoma was the most important feature [5], so this will be a necessary part of the final product.

δ. The second most important feature was the detection of melanoma from pictures, and a significant amount of people also said it was the most important feature of the app [5]. Therefore, the diagnosis of melanoma from pictures will be important for the final app.

ε. The other features such as reminders, certainty of prediction, storage of photos and specialist contact were all voted for by a significant amount of people very evenly. Therefore, these are all important features that may come after γ and δ.

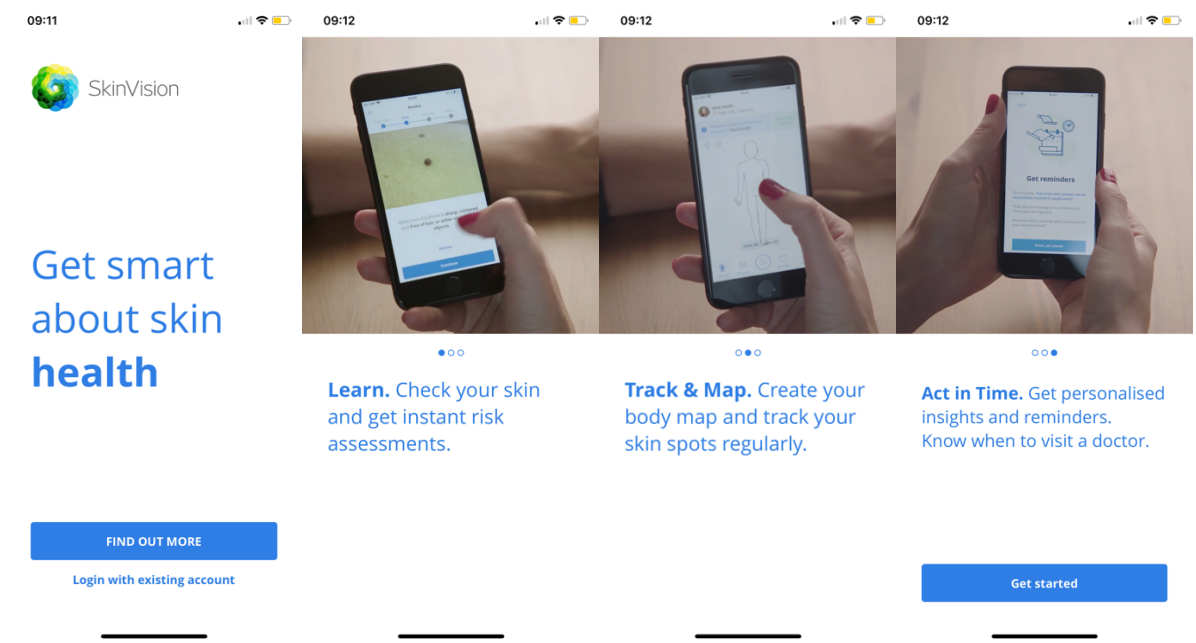
ζ. [6] showed that many people don't trust a computer's diagnosis very much, with an average trust of 6.2 on a scale of 1-10. Many people said that further backing by specialists, and evidence will help build trust [7]. As well as this an explanation of how the algorithm works will also help build trust [7], as someone who understands what is diagnosing them will have an easier time believing it.

d. Analysis of SkinVision

SkinVision is a premium, paid for medical service that helps you asses skin spots and moles for the most common types of skin cancer.

This app is the most popular for computer aided diagnosis of skin diseases (~ 1,800,000 users to date). It is downloaded for free from the app store and google play store, but most of the important features are only usable once you pay for the premium version. In this section I will go over the most important features of the app, explaining why it is the most successful in its field (which may end up influencing my design).

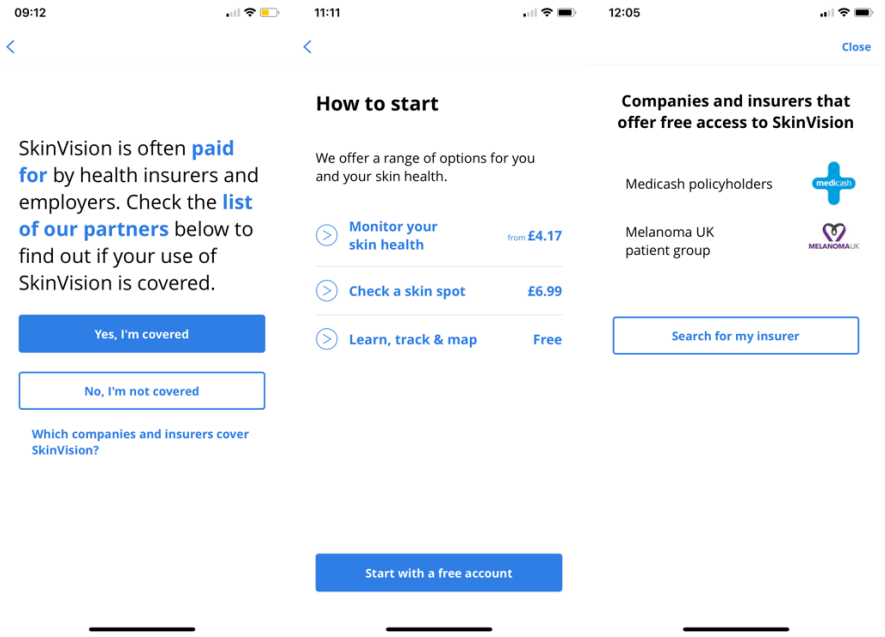
1. Opening Screen



These screenshots show what you see when you first open the SkinVision app. The app opens up with a button that allows you to understand what the app does in detail, through pictures and brief descriptions of the process. Then there is a button that leads you to the login screen.

I think the minimalist design of the opening screen help to make this app accessible to all, as everything is very easy to comprehend. The pictures and brief descriptions also help to further clarify what is happening in this app, and so make this app even more understandable.

2. Login Screen

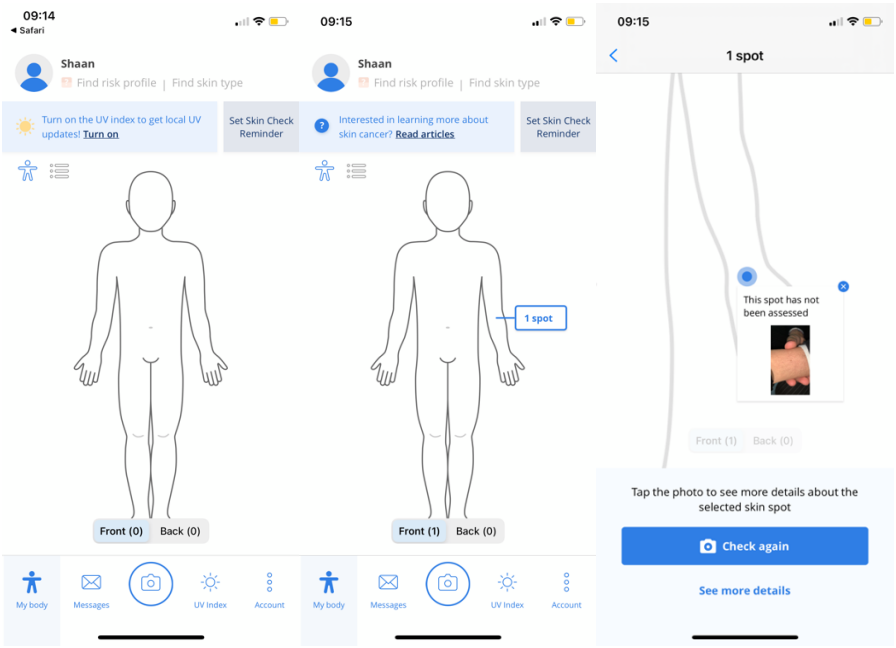


Once you click on 'Login' or 'Get Started' you are led to a page that explains that SkinVision is often covered by medicare and insurance companies. Then if you are covered you have to get some proof via that company to use the app, or you have to subscribe for different services. Unfortunately, so far there are only two insurers who offer free access to SkinVision.

Additionally, there are 3 subscriptions that you can use for the app. The first one is a subscription service, that starts from £4.17 per month, which allows you to scan your skin multiple times (the number depends on how much you pay for the subscription). The second subscription a one-time fee of £6.99 for detailed analysis of one picture of your skin. The final subscription is free, and allows you to take pictures and store them in the photo tracker, no analysis is conducted on the photo tracker.

Unfortunately, I do not think that the price is justified, as you could go to the doctors for free or a similar price and get a diagnosis that most people seem to trust more [c.ζ.]. The free subscription of the app seems to be beneficial for the user even though their skin does not get classified, providing many useful features such as the compilation of information and storage of skin photos which may turn out to be very useful.

3. Photo Tracker

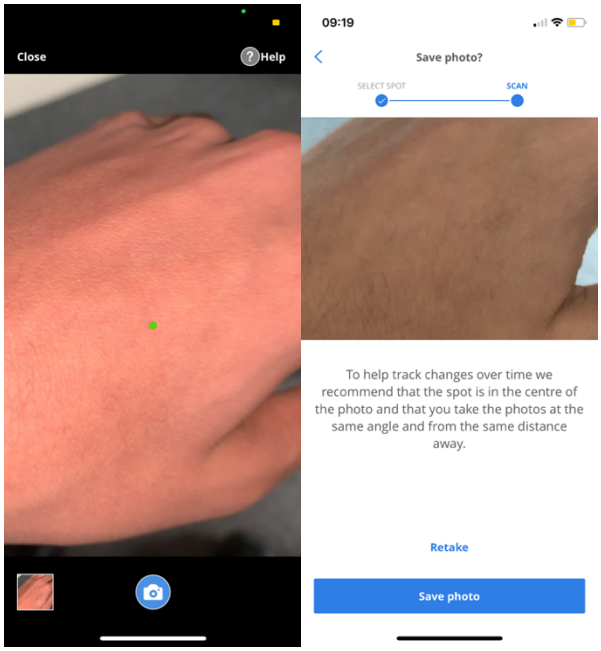


The “my body” page is the main page of the app. It is a visual representation of data that allows the data to be very easily accessed by the average user, essentially a photo tracker. This page allows you to click on different parts of the body and take a picture of that part of the body with the camera feature and store that photo. This visual representation makes it very easy to find the same photo later on for reference. This page also allows you to access every other part of the app.

I will be going through most of the features that are relevant to finding the best solution to this problem. I will not be going through features such as profile settings and messages because I do not believe they are necessary to help solve the problem of diagnosing melanoma.

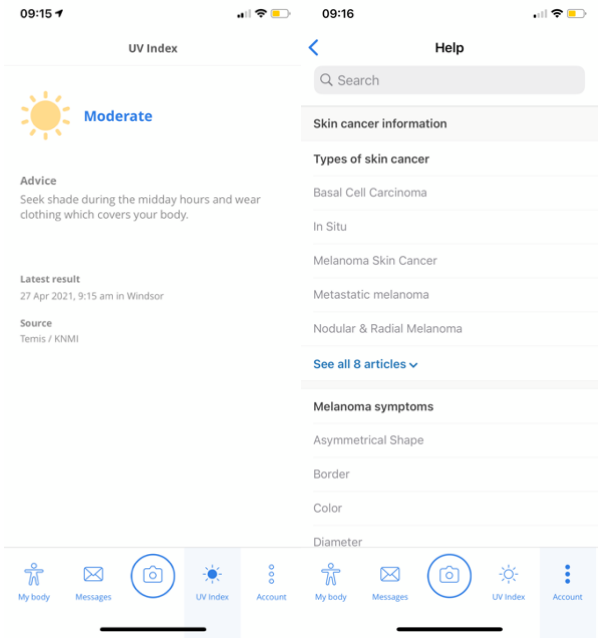
I think this main home page is great, but perhaps a bit too convoluted for a user who simply wants to diagnose possible melanoma, and you have to take a deep dive into the app to find information about self-diagnosis [c.γ.]. As well as this it is not clear whether the images you take are being associated with you at all (as you do have to make an account), and you are not made aware about whether your data is being further used for anything else or being given to anyone else [c.7.].

4. Camera feature



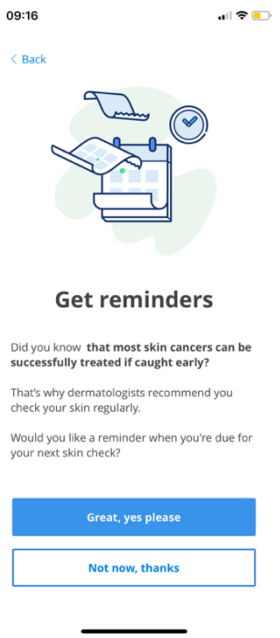
The camera icon on the photo tracker page allows you to access the camera feature of the app. This opens the phone camera, and automatically turns on the flash. The image you see has a green dot placed in the middle of the screen to help centre the melanoma, to make sure the image will be interpreted better by the algorithm. Having taken an image, you can retake it or save it to the app where it can be accessed later on. I think the idea of having a way to centre the melanoma and make sure the images are all of a certain size/distance from the skin will help a lot in increasing the accuracy of a prediction algorithm. The SkinVision app helps line up the mole to take a picture but does little to explain how far away the camera should be from the skin.

5. Information Displays



The app also has pages that display important information, such as UV level and skin cancer information. The UV Index page is very helpful for live information about risks of going outside, but the skin cancer information is not very easy to find (it is hidden deep in settings), and is not displayed very well.

6. Reminders



The SkinVision app also has an integrated reminders feature. This feature works by reminding you every so often to check your skin for any signs of cancer. I think this simple feature is very effective in getting people to use the app more often, potentially improving their health.

4. Proposed Solution

a. Choice of device

I have decided to make the product an app on an iPhone. There are many reasons for this, the most important being that it is a phone that is widely used that has a reliable camera with high resolution – something necessary for the app to function properly. The iPhone will also allow a lot of people to access the product via the App Store, which will allow more people to benefit from the app.

b. Success Criteria

In this section I will use all my findings from my research to detail the criteria that will make the final product a success. I will follow up each criterion with features that would meet it (in the following subsection). Following each criterion, I will also justify it, and provide a brief description of how the criterion would be tested. I will also leave evidence for justification in square brackets.

1. The product must comply with the law and current health guidelines

> *As this product would inherently have to deal with sensitive data, such as personal images or information, I will have to use current health guidelines and current law regulations to make sure the product does not infringe anyone's rights. I will have to look at data protection laws, specifically those that are about usage of personal data for medical reasons. [3.c.7.].*
>
> *Measure: Complies with current laws. (https://www.gov.uk/government/publications/code-of-conduct-for-data-driven-health-and-care-technology/initial-code-of-conduct-for-data-driven-health-and-care-technology).*

2. The main features of the product should be free and available to all

> *This product will be for the benefit of everyone, and so it should be available to as many people as possible.*

Measure: Is free to use for everyone with the correct hardware/software.

3. The product should be easy and intuitive to use

As many people will be using this app there will be a range of technology literacy of people using this app. Making this app as intuitive and easy to use will make this app accessible to all people – particularly helping the older demographic as there are a lot of people n that demographic that may find this product beneficial.

Measure: Feedback from stakeholders, complies with Jakob Nielson's heuristics.

4. The final product should be able to explain to the user how to self-diagnose melanoma and provide relevant information about self-diagnosis

Providing information for self-diagnosis for many people itself will help a lot of people with diagnosing melanoma. The survey conducted also showed that a lot of people wanted self diagnosis information in a product that would help diagnose melanoma [3.c.β.], [3.c.γ].

Measure: Check whether information in final product corresponds with current health guidelines; More than 90% of stakeholder feedback implies that there is enough clearly layer out information for self-diagnosis.

5. The final product should be able to aid in the process of diagnosing melanoma

α. The final product should be able to use images to aid in diagnosis.

Visual inspection is the main way of diagnosing melanoma today, and so it would be beneficial if the product could use images to aid in the process of diagnosing melanoma. As well as this, my research thus far suggests that a diagnosis without the use of images would not be reliable enough. [3.a.γ], [3.a.ε.], [3.c.δ.].

Measure: Checking if the usage of image data assists in the diagnosis of melanoma (accuracy of algorithm > 80%).

β. The final product should take into account other risk factors for its diagnosis.

Diagnosing melanoma purely from images may seem to be quite accurate, but, as I learnt from my research, taking into account other risk factors such as skin type and family history will help make the app even more reliable and accurate. [3.a.β.].

Measure: Checking if the app accounts for factors other than images when making a diagnosis.

γ. The final product should provide a certainty in its diagnosis

It is important to provide a certainty in prediction to let the user know that the final product may not be sure about its prediction so it is better not to take what it says for granted, but instead as advice about whether the user should go see a specialist. [3.a.β.], [3.c.ε.].

Measure: Checking if the app provides a reasonable certainty when making predictions (certainty should be above 80% for test data).

6. The final product should be able to store images of the user's skin in an accessible way

Storing the images in an accessible way will allow the user to check their skin for any evolution over time – something that is a strong indicator of melanoma. This will also allow the user to access their photos later on should there be a need. [3.c.β.], [3.c.ε.].

Measure: Feedback from stakeholders.

7. The final product should be able to take images of the user's skin in a way that they are all consistent.

> *For an algorithm to make an accurate prediction about melanoma from an image, all the images need to be of the same shape and size, and the suspected melanoma needs to be in a similar location in the images. Therefore, the product needs to be able to help the user take images in a way that the product can make beneficial use of. [3.d.4], [3.a.γ.].*
>
> *Measure: Inspection to ensure all of the images reach a certain level of similarity.*

8. The final product should not take up too much space in storage

> *The product should not take up too much storage space so that it can be stored on most devices regardless of their storage capacity.*
>
> *Measure: Final product should take up less than 100 MB in memory.*

c. Features

I will label each feature with a priority, with [E] representing an essential feature, [U] representing a useful feature that is not essential for the app, and [A] representing an auxiliary feature that is not necessary at all for the app, but it would be beneficial in some way. I will prioritise creating an app with all the [E] features, and then go onto [U] and [A] if I have enough time.

1. Information page. [E]

> *The final product should have a page that displays all the relevant, up-to date information about melanoma. This page should include information about topics such as self-diagnosis (ie symptoms, risk factors), information about disease and what to do if you think you have it.*
>
> *Justification: [b.4.]*

2. Functional image storage. [E]

> *The final product should be able to store images in a functional way, allowing users to access those images with ease and intuitively. A user should be able to see images of similar areas in a chronological order so that diagnosis can be made easier.*
>
> *Justification: [b.6.]*

3. Integrated camera access. [E]

> *The final product should have integrated camera access that helps the user take images of the skin in a way that will help enhance both human and algorithmic diagnosis of melanoma. This camera should store the images to the functional image storage.*
>
> *Justification: [b.7.]*

4. Classification of skin using images. [E]

> *The final product should be able to make predictions about whether the user has melanoma or not based on images taken by the integrated camera (so that they are all in a similar format).*
>
> *Justification: [b.5.α.]*

5. Certainty of prediction. [U]

> *The final product should provide the certainty of its prediction whenever it makes a prediction about the user.*
>
> *Justification: [b.5.γ.]*

6. Likelihood of melanoma from other risk factors. [U]

The final product should be able to take into consideration other risk factors, to help make the prediction/certainty even more accurate. The risk factors it should take into consideration are: Ultra Violet light exposure, Amount and type of moles, Type of skin, Family history of melanoma, Personal history of melanoma, Immuno-comprimisation, Age.

Justification: [b.5.β.]

7. Reminders to check skin. [A]

The final product should provide reminders to the user to check their skin every month.

Justification: from my research I found that melanoma is a very aggressive cancer and so it must be dealt with quickly [3.a.α.], and so it is necessary to diagnose melanoma at an early stage. Providing reminders would make more people check their skin more often, allowing for more melanoma to be caught early, which can be dealt with more easily than melanoma caught late.

8. Current skin cancer risk conditions in local area. [A]

The final product should provide live information about weather conditions that may have associated cancer risks. Such weather conditions include UV exposure, pollutant exposure, acidic skin irritant exposure.

Justification: if people are made aware of the risks involved with going outside on certain days, they may lower their risks by avoiding going out on high-risk days. This is particularly helpful for the older demographic who are more likely to get affected by melanoma.

d. Hardware Requirements

1. The device that the final product runs on must be an iOS device

This is because the IDE I am using to build my project only compiles on iOS devices.

2a. Supported devices must have high quality cameras
2b. The device must be an iPhone X or a newer generation of iPhone to be supported

This is due to the fact that the older generations of iPhones do not have cameras with high enough resolution to take pictures for predictions about melanoma. As well as this the algorithm that carries out predictions may be very computationally taxing, so newer iPhones would perform better.

3. The device must have around 100MB of memory available

This is because the machine learning models to make predictions may be quite large. As well as this the images stored will be high quality so they will also require a lot of memory space.

4. Computer hardware must be good enough to allow for training of models on thousands of images

The computer that the final product is created on must have capabilities to train a machine learning algorithm on thousands of images in a reasonable amount of time so that I can try many different models and use the one with the best accuracy.

e. Software Requirements

1. The development computer should be running macOS 10.14 or later

I will need macOS 14.0 or later as I will be using swift's createML to help me build my machine learning models.

2. The device should be running iOS 14.0 or higher

This is because many of the features for the app's user interface will be using SwiftUI, which is only properly supported after iOS 14.0.

f. General Limitations

1. Time Constraints

As I will have to do this project as an A-level project alongside my other A-levels I will not have too much time to work on this project. Therefore, I may not be able to implement all the features of the proposed final product.

2. Distribution Limitations

As only around 50% of people in the UK have iPhones, not everyone will have access to this app. As well as this not everyone will have a new enough iPhone to make use of the image prediction feature.

g. Software Limitations

1. Create ML framework

As the apple ecosystem does not allow 3rd party machine learning frameworks to be used very easily, I will have to use createML. This will mean that I will not have complete control of every parameter in the machine learning algorithm.

2. Support for older iOS versions

The final product will require iOS 14.0 or above so any devices with older software will likely not be able to use the product.

3. User Interface on different devices

Due to the many different sizes of iOS devices, I will not be able to make the app perfectly optimised for every device. SwiftUI does automatically format apps for every device, so hopefully that will be sufficient.

h. Hardware Limitations

1. Camera Resolution

Unfortunately, iPhone cameras are unable to take dermascopic pictures, and can only go up to a certain resolution. This may cause the images to not contain clear enough information for accurate predictions.

2. Device Storage

As mentioned earlier the iPhone will require at least 100MB of storage available for the product.

3. Device Type

As mentioned earlier the app will only be able to run on certain iOS devices, and so will not be accessible by every single person.

4. Testing on devices

As there are so many devices that this product can run on it will be hard to test the app on each and every one of these devices. I will only be able to test the device on an iPhone X or iPhone 12 Pro.

[ref_1] - Global cancer statistics 2020: GLOBOCAN estimates of incidence and mortality worldwide for 36 cancers in 185 countries
https://acsjournals.onlinelibrary.wiley.com/doi/full/10.3322/caac.21660

[ref_2] - How accurate is visual inspection of skin lesions with the naked eye for diagnosis of melanoma in adults https://www.cochrane.org/CD013194/SKIN_how-accurate-visual-inspection-skin-lesions-naked-eye-diagnosis-melanoma-adults

[ref_3] – DermNet NZ https://dermnetnz.org

[ref_4] – The International Skin Imaging Collaboration https://www.isic-archive.com

Design

1. Decomposition

a. Problem Decomposition

In section 1.b. I have inserted a structure diagram that shows how the final problem can be decomposed into smaller problems. The black arrows represent the breakdown of a feature into smaller problems so that the feature is easier to understand. The green arrows show what features require data/information from other features (ie the functional image storage requires the camera feature of the app to work). The text at the bottom left of each feature shows which success criteria the feature helps fulfil the most (which are mentioned in subsection 4.b. of the Analysis section). Some of the success criteria such as the ease of use [3] and following health guidelines [1] do not fit into the structure chart and will be addressed elsewhere in the document.

I have broken the final product into three major parts that can be (mostly) tackled separately. In the following paragraphs I explain my reasoning for these choices, with anything inside square brackets ("[ ]") referring to success criteria from the analysis section.

Predictive Features

One of the most important features of this melanoma diagnosis app is the ability to help predict melanoma, and so diagnose it [5]. There are two major aspects of this feature – to provide a prediction from some kind of skin classification model, and also provide a certainty of prediction so that the user can understand whether the algorithm is sure about its prediction.

1. Skin Classifier

There are many approaches one could take to make a skin classifier, but by far the most accurate and effective approach is to use a machine learning model that can learn and understand what makes a melanoma a melanoma. A machine learning algorithm will be faster, and, sometimes, more accurate than a specialist therefore it makes sense to utilise machine learning for this application. Although there are many types of machine learning algorithms, they all follow the same general structure – collect training data, then train model, then based on the training performance optimise the model in an iterative method until the model reaches a certain level of performance. That is why I have broken the skin classification features into those three parts, which at the end of the iterative process will give the final classification model for the product.

This feature helps fulfil [5.α], [5.β].

2. Certainty of prediction

Along with a skin classifier the success criteria also mention that the final product should be able to provide a certainty in its diagnosis, and the method of providing a certainty is heavily reliant on which machine learning model is chosen and how it is implemented. This feature will not only rely on what certainty the machine learning learning model provides, but will also be influenced by the risk factors that the user may have.

This feature helps fulfil [5.γ]

Information Features

Another major feature of the melanoma app that arose from research in the analysis section was the ability to store and display important information regarding things that may help (both the user and the algorithm) diagnose melanoma. Furthermore some of these features are important for some of the predictive features to function properly.

3. Functional Image Storage

The analysis section showed that it will be necessary to have a page that stores images in a an accessible and functional way in the final product. This feature will have two main purposes: to store and display images in a user friendly way, and also to provide a database which the skin classifier can access and perform classifications on.

This feature helps fulfil [5.α], [6]

4. Other Risk Factors

In order to reduce the chances of a false positive or negative the final product will also need to take other risk factors into account. These factors will be given to the final product via user input. The main purpose of this feature will be to aid the skin classifier by using the risk factors to give a better certainty of prediction.

This feature helps fulfil [5.γ]

5. Self-diagnosis

This feature will provide the user with relevant, up-to date information about self diagnosis of melanoma. The feature will reference other parts of the final product such as risk factors and images stored to help make it easier for the user to self diagnose melanoma.

This feature helps fulfil [4]

6. Reminders

Reminders will server as a useful "extra" feature that will help the user remember to check their skin regularly to ensure that no melanoma gets to the stage where it is difficult to operate on, or even is untreatable.

This feature is one of the auxiliary features that will only be added if there is enough time. It is a helpful feature that is not necessary.

Hardware Features

7. Camera

The camera feature will allow users to take photos of melanoma in a consistent way. This feature will allow them to store their images on the final product and use it as input for the skin classifier. This feature has a sub feature "Camera guidelines" that will help the user align the suspected melanoma up in a way that allows the user to take consistent photos, allowing for better diagnosis from the skin classifier, as well as better photos to use by specialists in the functional image storage.

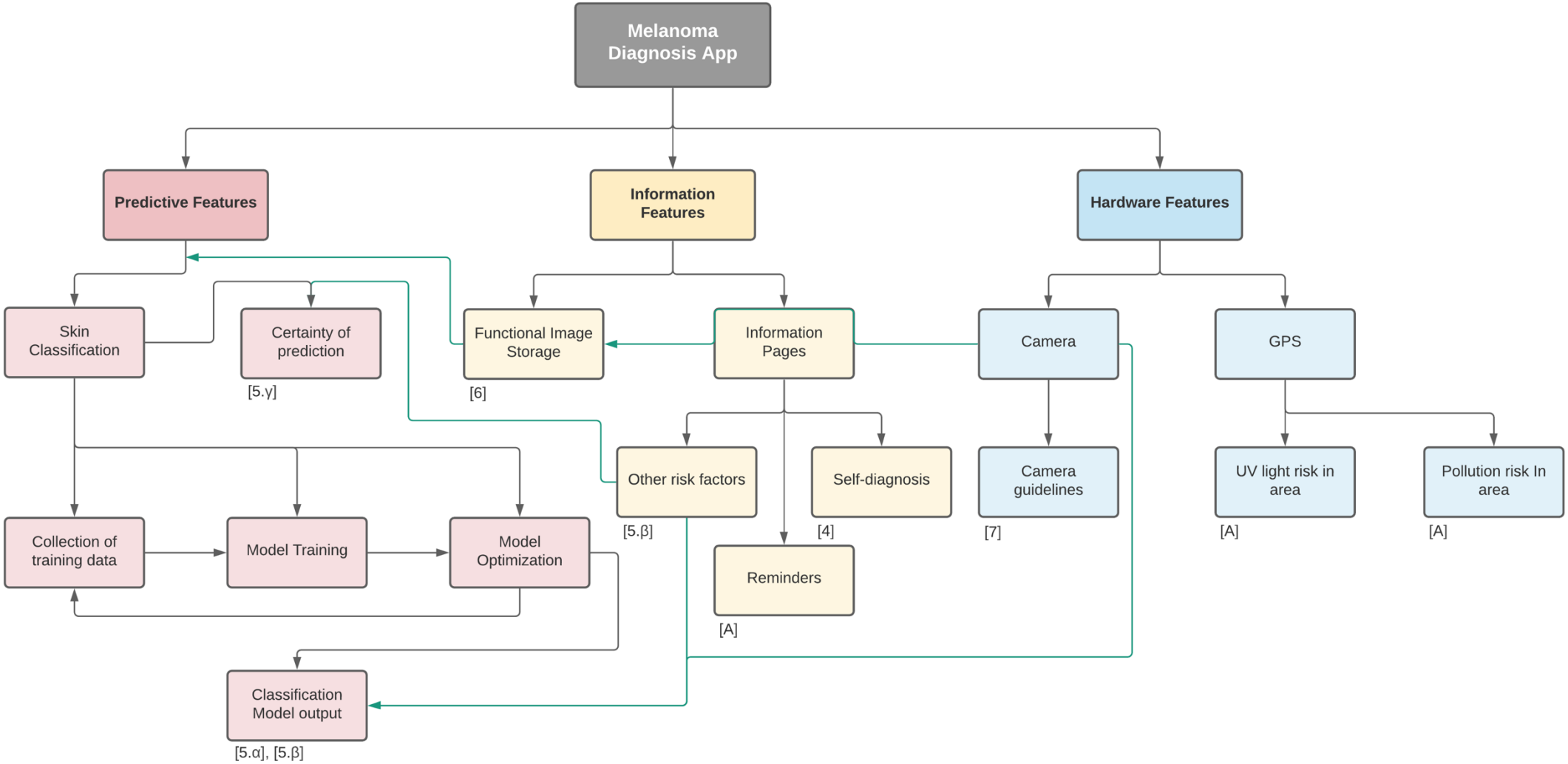
This feature helps fulfil [7]

8. UV light risk and pollution risk in area

My research in the analysis section showed that there are strong correlations between melanoma cases and exposure to UV light and certain pollution. Therefore, a feature that lets the user know about current risks from those factors will be useful.

This feature is one of the auxiliary features that will only be added if there is enough time. It is a helpful feature that is not necessary.

b. Structure Chart



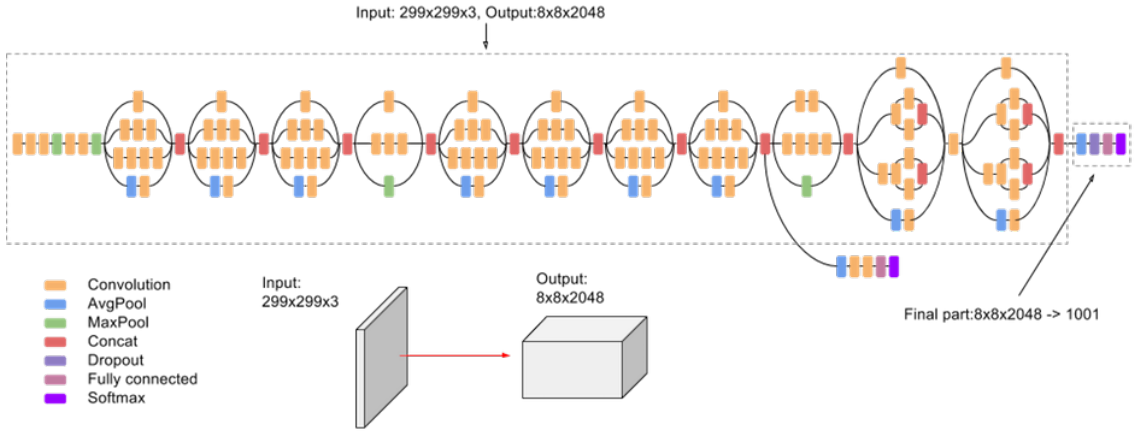
c. Algorithmic overview

In this section I will be giving an overview to how each of the features described in the previous two subsections will be implemented. I will show the algorithms in the form of flow charts in order to show the general flow of data and algorithmic complexity without limiting myself to a specific implementation (as it may turn out whilst programming the product that one of the parts of the algorithm is better with a slight adjustment).

1. Skin Classifier

The skin classifier will be a machine learning model that takes in an image of melanoma as an input, and classifies that image into one of two classes: melanoma or not melanoma. Through my research I have found many papers that offer different approaches to a similar problem, and I shall take previous research into consideration. A paper about skin cancer detection with deep learning by A.Esteva et al. (https://cs.stanford.edu/people/esteva/nature/) has a lot of information about using machine learning technology to recognise skin cancers, and so I will be using that as the basis for my algorithm in this section.

For the classifier I have chosen to use a convolutional neural network as the base architecture, research has shown these types of algorithms to be the most accurate for one-shot image classification tasks. Furthermore, the paper mentioned earlier also found that the Inception v3 network worked extremely well for this task (which is a variant of a convolutional network). Due to the nature of the problem I may have to test different models and evaluate their accuracy on unseen data, so until I actually create the models I will not be able to definitively say what works best. Below I have a diagram of a possible network that would work well.



As apple offers its create ML service, I will first attempt to use their architecture that apple claims "automatically synthesises" to create an optimal model. This will likely use some variation of the blocks seen in the key in the bottom right of the image. If the create ML model is not sufficiently accurate, I will create a hand-built model based off of the inception v3 architecture that has been shown to work well in a variety of vision tasks. Below I will briefly describe what each block in the key does.

A convolution refers to a downsizing of inputs usually in the form of 3 dimensional matrices. A filter of a certain dimension (ie 3x3) is applied to the layer and what it does is it extracts the most important features from that 3x3 filter to take into the next layer. This filter is applied in an iterative fashion over every part of the input matrix (like a cloth covering each part of a window whilst cleaning), and so an output is produced with a different size, containing only the important features of the input. These convolutions have to be trained via back-propagation (which requires a lot of data) in order to know which features are important. These will almost definitely be the backbone to the final model.

AvgPool uses a filter in the same way as a convolution, but instead takes the average of all items inside the filter, creating an output with a much smaller size. This is essentially an untrainable convolution layer that is used to slim down the input size in order to speed up the algorithm (by having smaller data to compute).

MaxPool has the same structure as AvgPool, but instead of average all values it takes the maximum of all values inside the filter as part of the next layer.

A concat layer is a layer is simply a layer that takes multiple inputs (many matrices in this case), and concatenates them together into one matrix (or list). This happens often in the algorithm to make sure that all the different blocks are working cohesively, and are not producing independent predictions.

The dropout layer is used for regularisation, which is essentially making sure a model doesn't memorise the inputs corresponding outputs rather than actually learning what features make melanoma melanoma. This works by randomly turning certain blocks "off", and so not considering their input to the final prediction at all. This makes sure that the algorithm does not depend too heavily on certain blocks, and can come to correct conclusions even if certain blocks do not function as expected.

A fully connected layer refers to a layer of many neurons that perform linear mappings. They convert matrices and the convoluted inputs into scalar information that can be used by the final soft ax layer to make a prediction.

The final softmax layer is just a function that takes inputs and converts them to a value between 0 & 1. This layer's output will be used to provide a certainty in the final output as well as provide a prediction for the user.

The machine learning model I will be using will be trained as a supervised algorithm. This means I will be training the machine learning model using labelled data, and using that labelled data to evaluate the model's performance every few iterations of learning. This evaluation will be given in the form of a loss function. In the case of classification, a categorical cross-entropy function works extremely well (shown below).

$$\begin{aligned} J(\theta) &= \frac{1}{N}\sum_{n=1}^{N} H(\hat{P}_{data}(\cdot|x^{(n)}), P_{model}(\cdot|x^{(n)})) \\ &= -\frac{1}{N}\sum_{n=1}^{N}(1-y^{(n)})\log(1-\hat{y}^{(n)}) + y^{(n)}\log\hat{y}^{(n)} \end{aligned}$$

Where y is the true label of the image (in this case 1 if image has melanoma and 0 if not), and y hat is the output from the softmax layer of the model (a number between 0 and 1). As you can infer from the maths if y and y hat are not similar J (the cost function) will be large, if they are similar then J will tend to zero. The goal of this model will be to minimise this cost function by changing the model's parameters.

This minimisation of the cost function will be achieved through back propagation, which is finding the derivative of the cost function and then propagating backwards through the network with derivatives in order to find which way the parameters should change so that the algorithm has a lower loss. The back propagation step will be different for each layer; therefore, I cannot give a general formula.

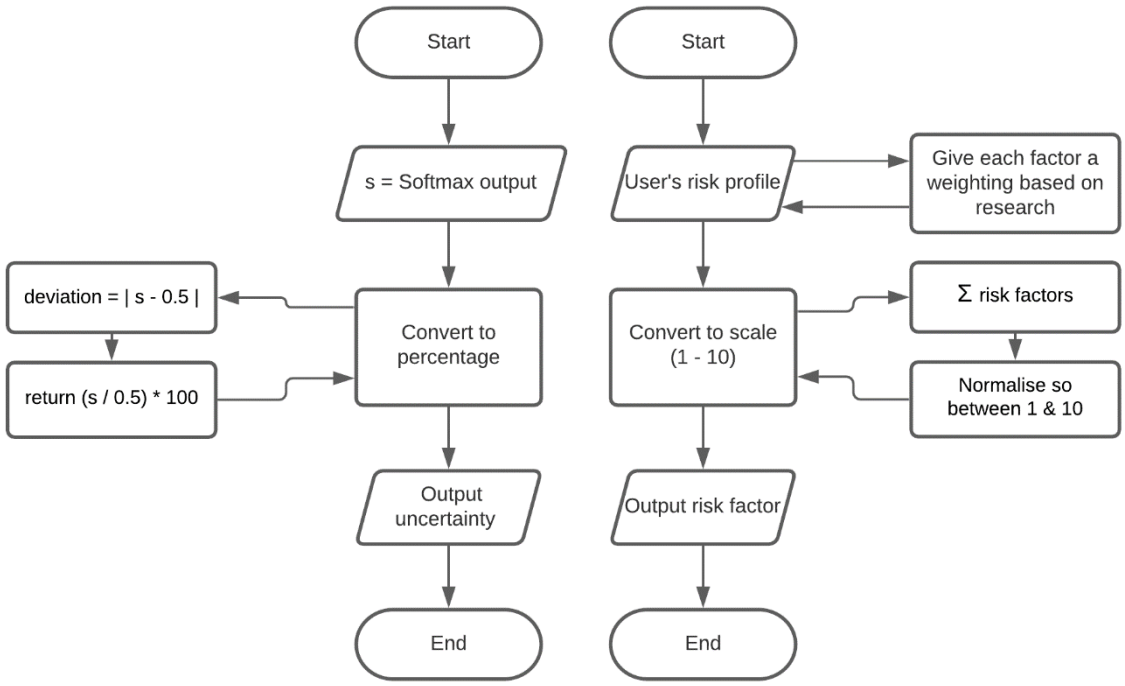
The algorithm that I will use to optimise the back propagation and training aspect of this model will be the Adam optimisation algorithm. It uses a mixture of momentum and gradient descent to optimise the training time and make sure that the training algorithm does not get stuck at a local minima in the cost function (where a better model is possible but the training optimiser is unable to find it due to the gradients mathematically being stuck at 0).

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

This optimisation algorithm was originally developed by D.Kingma and J.Ba, and the section above is taken from their paper "Adam: a Method for Stochastic Optimisation". It provides the update rules for the previous layers, as well as some hyper parameter values that they found worked very well for a wide range of tasks.

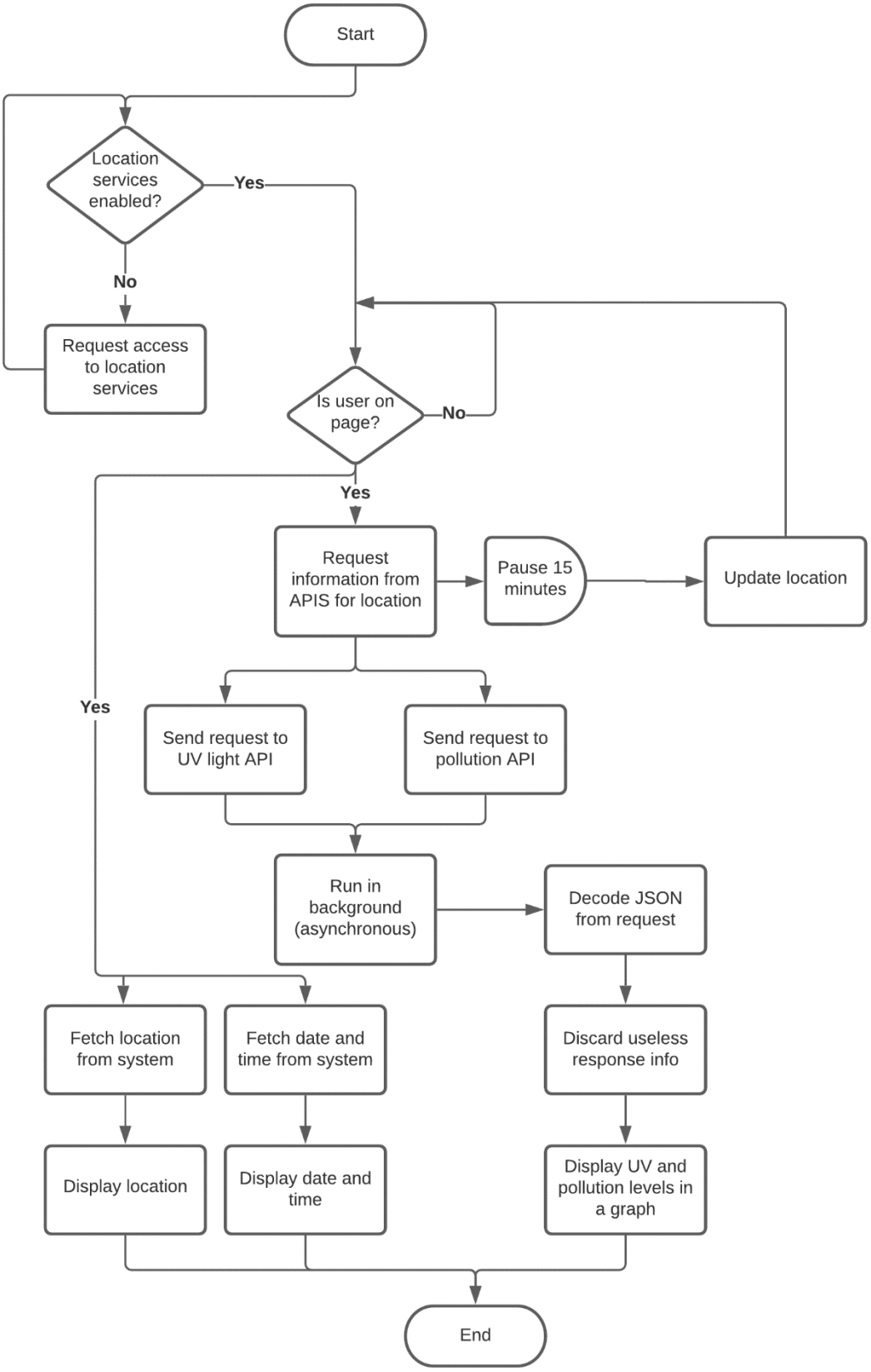
2. Certainty of Prediction



The certainty of prediction feature will be based off of the output from the classifier. The softmax output will give an output between `0` and `1`, and so the certainty will be the deviation of the output from `0.5`. This will be then converted into a percentage to display to the user.

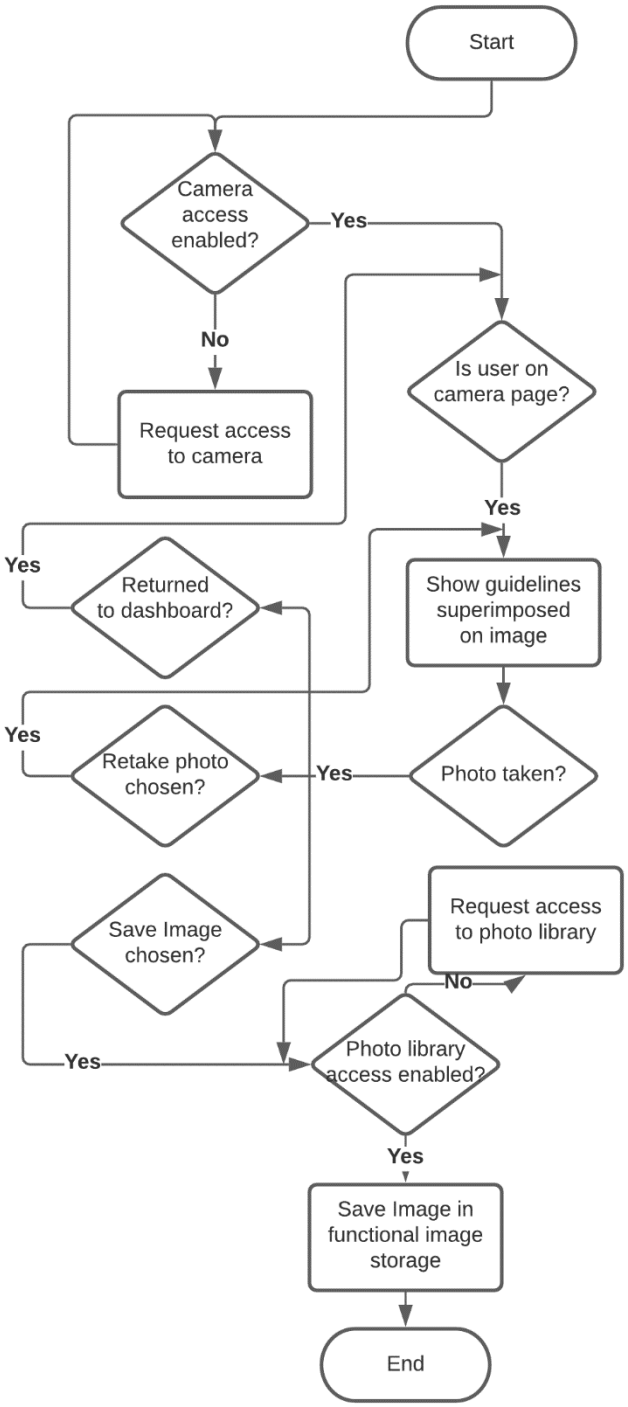
I have also included the risk algorithm here. The user's risk profile will be taken in as the input, and a linear function will be applied. Every factor in the risk profile will be given a weighting (between `0` and `1` based on evidence from research - https://www.cancerresearchuk.org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors). The factors will then be summed up and normalised to give a risk factor score between zero and ten, which acts as an arbitrary scale so that the user can have more information about the certainty of their diagnosis.

3. UV light and pollution risk in area



This algorithm just asks the user for location services access in order to send requests to APIs for information about melanoma risks for current UV light and pollution levels.

4. Camera



This flowchart shows how a user will navigate through the camera feature. It shows all the processes that may happen when the user is trying to use the camera, including any access that may be required whilst using the camera feature for the first time.

2. Solution Mockup

a. Usability Measures

In this section I will create a Balsamiq wireframe that will show the UI structure of the app. It will not accurately depict the aesthetics of the app, but will provide a good representation of its final structure. To help guide my design I will use Jakob Nielson's 10 general principles for interaction design, this will allow me to create a very usable app (helping fulfil [3])

Jakob Nielson's 10 general principles are as follows –
(www.nngroup.com/articles/ten-usability-heuristics/)

1 Visibility of System Status

Designs should keep users informed about what is going on, through appropriate, timely feedback.

2 Match between System and the Real World

The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon.

3 User Control and Freedom

Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted state.

4 Consistency and Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

5 Error Prevention

Good error messages are important, but the best designs prevent problems from occurring in the first place.

6 Recognition Rather Than Recall

Minimise the user's memory load by making elements, actions, and options visible. Avoid making users remember information.

7 Flexibility and Efficiency of Use

Shortcuts — hidden from novice users — may speed up the interaction for the expert user.

8 Aesthetic and Minimalist Design

Interfaces should not contain information which is irrelevant. Every extra unit of information in an interface competes with the relevant units of information.

9 Recognise, Diagnose, and Recover from Errors

Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

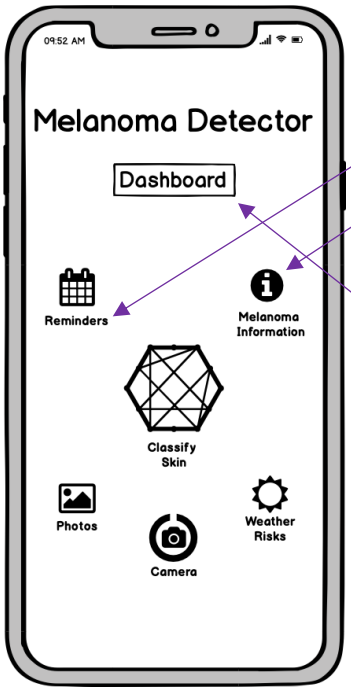
10 Help and Documentation

It's best if the design doesn't need any additional explanation. However, it may be necessary to provide documentation to help users understand how to complete their tasks.

In the following section I will show an initial mockup of the final product, and reference which heuristics are explicitly being used in each view. Unfortunately, I will not be able to properly develop documentation for the app given the limited time, so I will use this document as the documentation for the user (helping fulfil [10]). The numbers in the following section will be referring to the heuristics described above.

b. Balsamiq Mockup

1. Dashboard

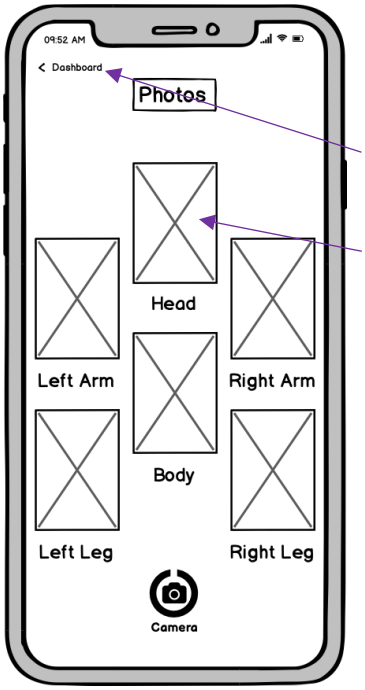


The dashboard will be the main view in the final product. I have opted for a very minimal design for the dashboard in order to not have a home screen that is too overwhelming for users [8].

The brief descriptions under each button also provide information for the user to understand what each button does [2], but after a while the user can familiarise themselves with the symbols for different features, and so make the user experience much more efficient [6] [7].

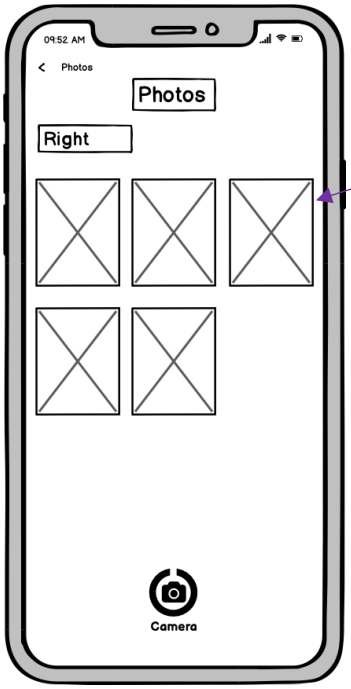
The text box below the app name helps inform the user of their current location on the app wherever they are, allowing for a more user-friendly experience [1].

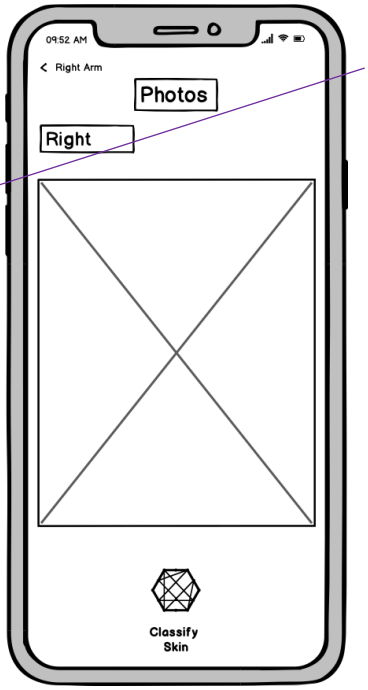
2. Functional Image Storage



Once the user clicks on the photos button, they will be led to a view that stores the user's images in a functional way. The images are arranged in a way that they correspond with the physical location of the human's anatomy. Once you click any button there will always be an exit button on the top left of the app that will lead to the dashboard, making sure users do not get lost in the app [3].

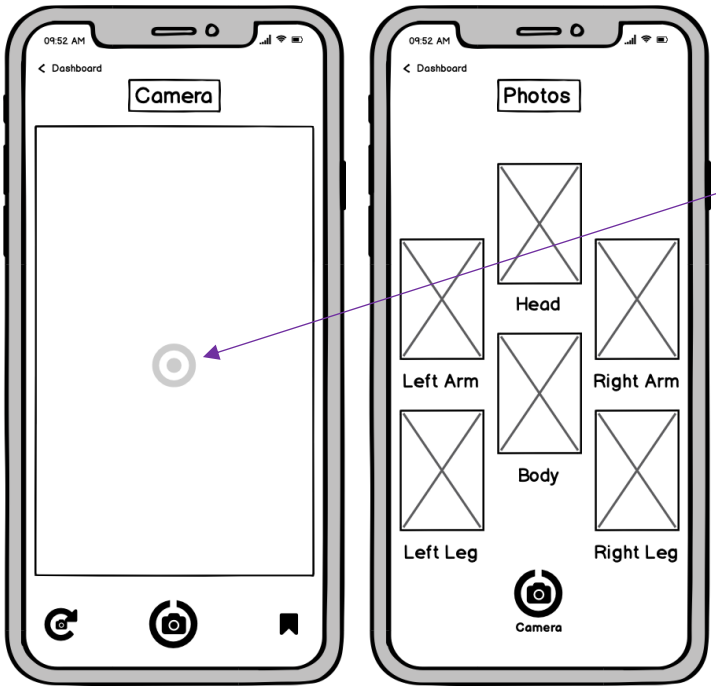
Each group can be clicked to show all the photos relating to that location, allowing for a more minimalist design that doesn't have too much going on in one view [8].





When the user clicks on one of the groups, they will be led to a screen showing all the photos they have taken in that area in chronological order. Here they can look at their old images, or take a new one to store there [6]. If they click on one of the photos a full resolution version of it will be shown to the user, so that they or a specialist can use it for diagnosis. As well as this having opened up a full resolution image the user can click on a classify skin button that appears to use that image as input for the skin classifier [6].

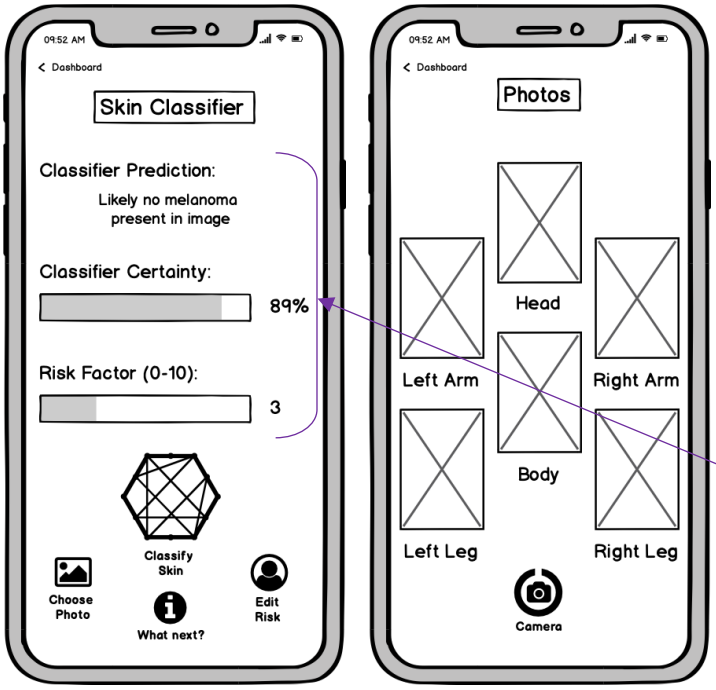
3. Camera



If the user clicks on the camera app, they will be asked for permission to access the camera [1]. The camera interface will be very similar to modern day smartphone interfaces [2] [4], but will have an overlay that helps the user take consistent photos.

Once the user has taken a picture, they have the option to either retake it (bottom right symbol) [3] [5], exit the camera (top right) [3], or save the image (bottom right). If the user chooses to save the image, they get led to the photos view where they choose which group to save it under.

4. Skin Classifier



The skin classifier can be accessed either by clicking on the button on the dashboard or by navigating to an image in the photos view and then choosing an image to use [6] [7]. If the user has not already selected an image they have to do so via the choose photo button on the page. They can also edit their risk factors from this page, as well as get information about what to do next once the classifier makes a prediction. The large classify skin button will run the computer vision algorithm on the image chosen. [1] [2].

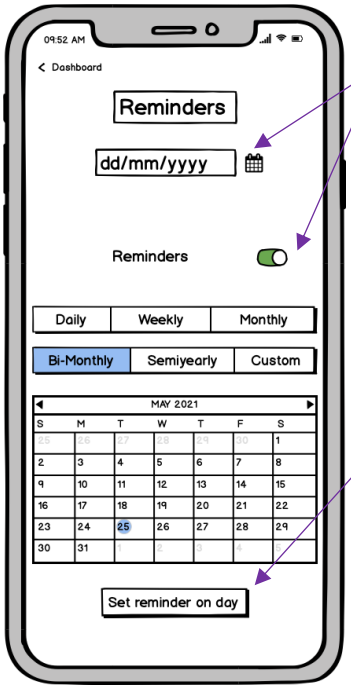
The page shows three main pieces of information. The first is what the classifier predicts given the inputs, the second is the certainty of that prediction and the final one is the user's risk factor on a scale of one to ten (aiding the certainty feature). This minimal design should keep the user informed but not overwhelmed [8].

5. Information Page

The information page will display all the extra information the app has to offer. This includes information such as self-diagnosis, specialist contact and more information on the workings of the app (as my research showed that people would trust an app they understood more) [2]. This information will be shown in a list where users can click on items to learn more about them.

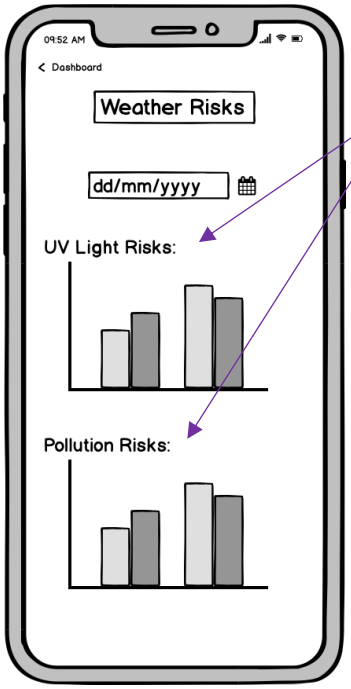
Users will also be able to edit their risk factors on this page, which will aid the classifier. It will ask a few very simple questions that can be edited any time.

6. Reminders



The reminders section of the app is an extra feature that will only be made if there is sufficient time remaining. It will show the current date and have a calendar [2]. It will allow users to enable and disable reminders, and set a frequency of reminders. It will also allow users to set reminders on specific days if they would like to.

7. Weather Risks



The weather risks part of the app will also only be made if there is sufficient time. It will also show the current date and time, and show the user two graphs. One will show UV light risk on the y axis (in arbitrary units [2]), against time on the x axis. The other graph will show pollution risk (in arbitrary units [2]) against time on the x axis. The user will receive this information based on their current location, and so will be asked to enable location services for this app whilst the using it.

c. Stakeholder feedback

I asked Dr Lyman for his thoughts on the design section so far, and below is his response.

"My only concern would be the risk algorithm, as each of the risk factors does not account for 10% of the risk for developing melanoma. That could lead to people being falsely reassured or worried. Perhaps use the same risk factors but categorise it into slightly more ambiguous categories ('lower risk', 'medium risk', 'higher risk'). I also don't think that the risk factor should be shown on the 'classifier certainty' page, as someone's background risk of developing a melanoma is in unrelated to the immediate question of 'is this mole/spot a melanoma'? For example, if someone has a high percentage classifier but their general risk is low, they could feel falsely reassured.

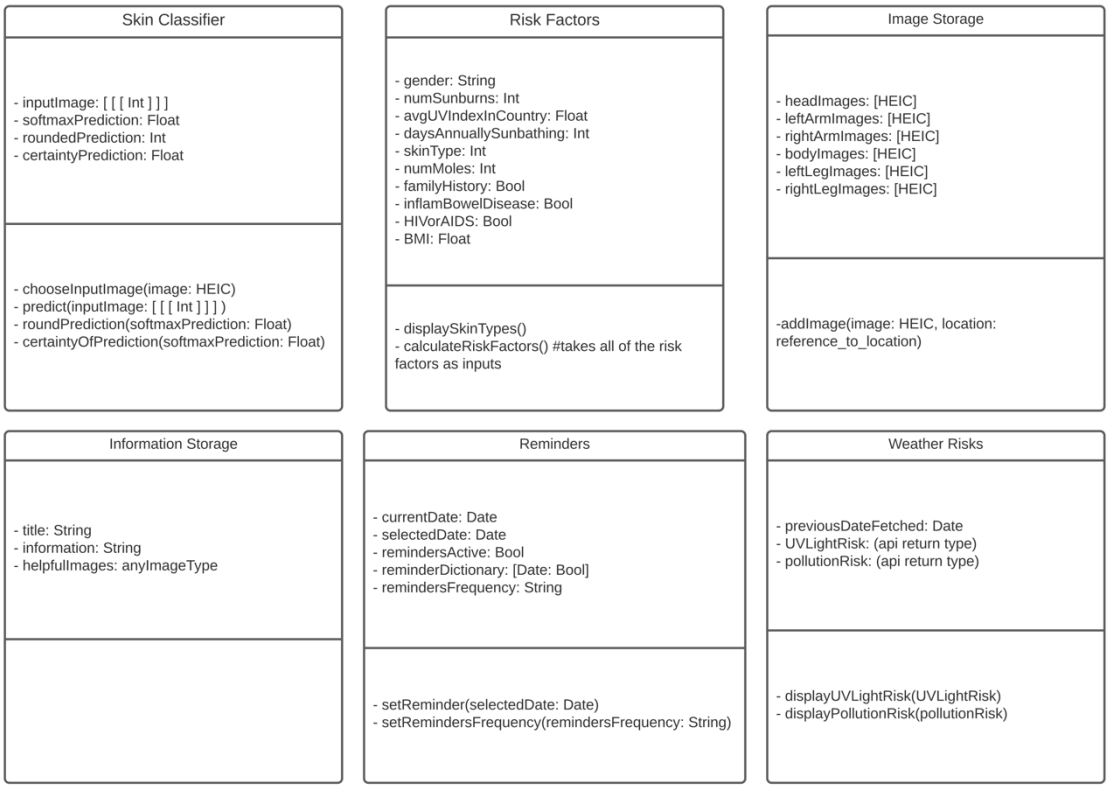
I think the interface looks really clear and clean. I think 'Classify Skin' could be made more user friendly, like 'Scan your skin'. Also, perhaps there should be a security element to viewing your photos (e.g. fingerprint/password) as this is sensitive info. And I like the 'reminders' section. This could be for both an annual skin check for healthy people, and monitoring a suspicious mole over the course of weeks to see if it changes/grows.

I really like the UV light risk feature...from what I've seen I'm guessing this will take into account time of day? The evidence suggests UV risk (in the Northern Hemisphere at least) is highest between 10am and 4pm.

Hope this helps a little. It's really good! "

3. App Data Structure

a. UML class diagrams



The class diagrams above break down my solution into individual classes I will create in Swift. These classes are designed to be reusable and to have as few dependencies as possible. Some of the classes such as Image Storage and Information Storage are there as classes to help organise the data stored by the app. The Reminders and Weather risks classes use APIs to get relevant information and display it. The Skin Classifier class will use data stored in other classes in order to make predictions. The Risk Factors class will store information about the user in order to calculate a risk factor value.

b. Validation

To maintain the integrity of my app and prevent any crashes/unexpected data in functions I should consider all the possible places that the user may enter input data and create validation rules. This will help ensure that there are no unexpected errors in the app that make the user's experience undesirable.

View	Type of Check	Data Checked	Justification
Camera	Presence	Access to Camera	The user must allow access to the camera in order for the app to take photos.
	Presence	Image taken	The user must ensure that the image taken is of the human skin, otherwise the app will not give a result that can be interpreted.
	Data Size	Image taken	The user must ensure that the image taken is not blurred, and is high enough resolution to be used by the classifier.

Skin Classifier	Presence	Input Image	The user must make sure that they have chosen an image to use as the input for the skin classifier.
	Presence	Risk Factors	The user must make sure that they have entered in their risk factors to use for calculating their risk factors.
Information	Presence	Risk Factors	The user must make sure that they have entered their risk factors to the best of their knowledge.
	Data Type	Risk Factors	The user must make sure that the data type for each risk factor entered is correct. Each risk factor has to have the correct data type entered as input.
Reminders	Data Type	Selected Date	The selected date entered must be a valid date.
	Data Type	Reminder Dictionary	The dates in the reminder dictionary must be valid dates.
Weather Risks	Data Type	UV light API data	The data returned from the API call must be converted into usable data types. Also need to check that the API call has returned data, and need to validate correct data returned.
	Data Type	Pollution API data	The data returned from the API call must be converted into usable data types. Also need to check that the API call has returned data, and need to validate correct data returned.

4. Testing plan

a. Testing during development

View	Test description	Example test data	Expected	Justification
Camera	Take a picture of human skin **Valid**	Image of back of hand	Popup to ensure user has taken a picture of skin, which then leads to image being saved.	User must be able to save an image if they have taken a picture of skin.
	Take a picture that does not include human skin **Erroneous**	Image of a chair	Popup to ensure user has taken a picture of skin, which then leads to image being deleted.	Images not of human skin should be rejected and not saved.
Skin Classifier	Calculate risk factor without entering risk factors **Erroneous**	Not all risk factors entered	Error message showing that risk factors not entered correctly.	The risk factor cannot be calculated without the necessary information.
	Input an image known to have melanoma **Valid**	Melanoma positive image	Classifier predicts there is melanoma in image.	The classifier must predict that there is melanoma in image when there is.
	Input an image known not to have melanoma **Valid**	Melanoma negative image	Classifier predicts there is no melanoma in image.	The classifier must not predict presence of melanoma when there is none.
Risk Factors	Risk factors entered with correct data type **Valid**	Correct data type risk factors entered	Risk factor calculated correctly.	Risk factor must be calculated if the entered values are correct.
	Risk factor fields left empty **Erroneous**	Empty field	Error message letting user know fields have been left empty.	Risk factor cannot be calculated if correct data not present.
	Risk factor fields entered with incorrect data type **Erroneous**	String data in integer data field	Error message letting user know that incorrect data type entered.	Risk factor cannot be calculated if correct data not present.
	Risk factor fields entered with extreme data **Erroneous**	Age = 9999	Error message letting user know that the data entered is not reasonable.	Risk factor cannot be calculated if incorrect data is present.

Reminders	Reminder chosen for valid date **Valid**	Chosen date = 01/01/2023	Reminder set for chosen date.	If a valid date is chosen set a reminder for the date
	Reminder chosen for invalid date **Erroneous**	Chosen date = 29/02/2023	Error showing date chosen is invalid	If invalid date is chosen do not set a reminder.

b. Post development test plan

There are a few important things that I will test once I have finished building this product. The first is checking whether the app takes up less than 100MB of storage (success criteria 8), and ensuring that the final product is free for everyone. I will also need to ensure that the app follows the latest health guidelines.

To make sure that the product is easy and intuitive to use (success criteria 3), I will be giving the final product to stakeholders to test. I will check whether it is easy for them to use any feature that they would like to, and whether they can do that intuitively. In order to further improve the usability of my app I will take feedback from the stakeholder about the design of the app and how it can be made better. I will then incorporate these features into the app.

Development

1.Ticketing

a. Software used

I decided to use Microsoft Planner to do my ticketing for this project. It allows me to create multiple buckets in which I can add tasks I need to do. I decided to create the following buckets –

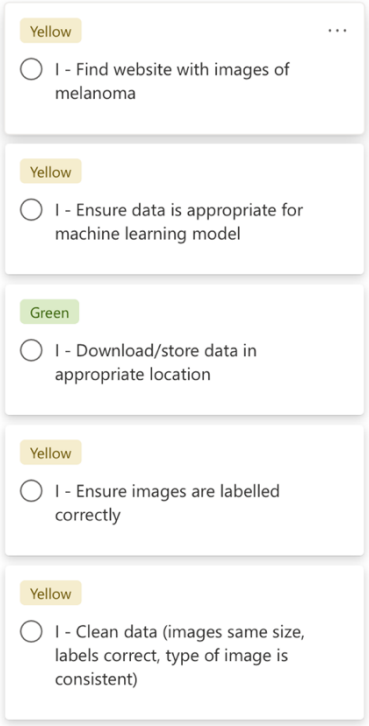
1. Sprint Backlog
2. Current Sprint
3. Finished Sprints

The reason for these buckets is so I can organise my sprints. The Sprint Backlog contains all my sprints that I need to do. The Current Sprint represents the set of tickets I am currently working on. Each of my tickets has a number next to it that refers to what sprint that ticket is in, this is just to help organise all my tickets into distinct sprints whilst using Microsoft planner.

Furthermore, there is a colour next to each ticket and sprint, which represents how difficult I expect each ticket to be. This will help me manage time and know what will require more effort to implement (green = easy, yellow = difficult, red = will require significant learning, cranberry = even more time consuming than red).

2. Sprint 1 – Collection of Melanoma Data

a. Ticket overview



b. Ticket 1 - Find website with images of melanoma

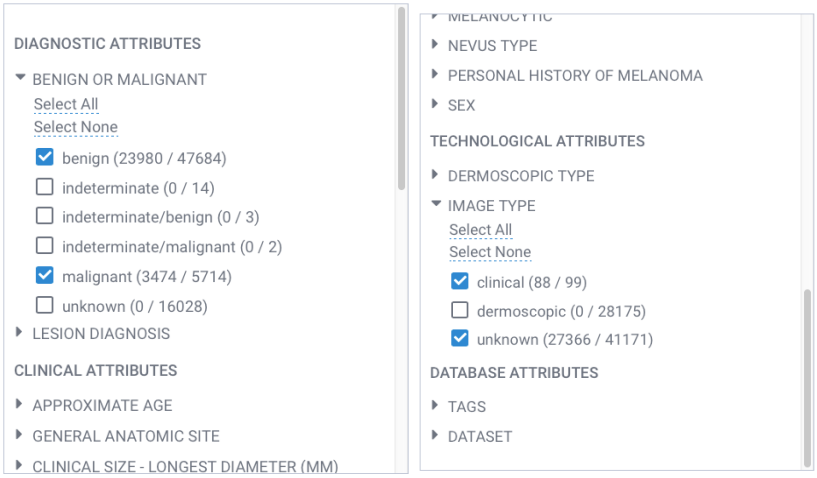
Through conversations with stakeholders, discovered the International Skin Imaging Collaboration (ISIC). ISIC (https://www.isic-archive.com/) contains images of benign and malignant melanoma used to train dermatologists to detect melanoma. Therefore, the data can also be used to train a machine

learning model that detects melanoma. There were also enough images so that a model can be trained sufficiently accurately.

c. Ticket 2 - Ensure data is appropriate for machine learning model

The ISIC website contains tens of thousands of images of melanoma, all taken differently. I had to ensure that none of the images I used were taken using microscopes or high-resolution dermascopes as those images would not be similar to ones that can be taken by iPhones. I also had to ensure that the data was not biased and I did not use the same image multiple times, or only images taken by a single study/organisation. I also had to ensure that all the images were higher resolution than 299x299, as that is the minimum resolution, I could use with swift's create ML feature to create an image classifier.

Fortunately, the ISIC has inbuilt filtering features where I could remove all images that were too small or were taken by dermascopes. The feature is shown below in the screenshot. However, the ISIC did not remove all the dermascopic images so I had to sort through all of the data myself, selecting only those images that were clearly non dermascopic and could be taken by an iPhone camera.



d. Ticket 3 - Download/store data in appropriate location

I ended up finding around 500 malignant images that could be used for the machine learning model. As there were many more benign images than malignant, I did not use as many benign images that were at my disposal. This was to avoid bias in my machine learning model (which is where my model could constantly predict benign, and as there are many more benign than malignant images it will have a very high accuracy regardless of how good the actual model is). Therefore, I used around 500 benign images as well. I downloaded all these images locally to prepare for the next ticket.

e. Ticket 4 - Ensure the images are labelled correctly

Having downloaded all the melanoma images locally, I divided them into two sets – benign and malignant. Then these sets were further divided in a 9:1 ratio into training and testing sets. The majority of data in each set (around 450 images) were used for training the model, and the rest were used to test and evaluate the performance of the model. I called the latter, smaller set the testing set and the machine learning model never learnt from those images. This was so that I could test the model on these 'unseen' images, to ensure that it will be able to perform on unseen data rather than just the data it has learnt from.

f. Ticket 5 - Clean data (images same size, labels correct, type of image is consistent)

Fortunately, I did not have to clean much of the data as the ISIC had given images in the correct size and format whilst downloading. I just went through all of the data ensuring that there was nothing that would cause an error, and I found nothing that would cause an error.

g. Sprint Review

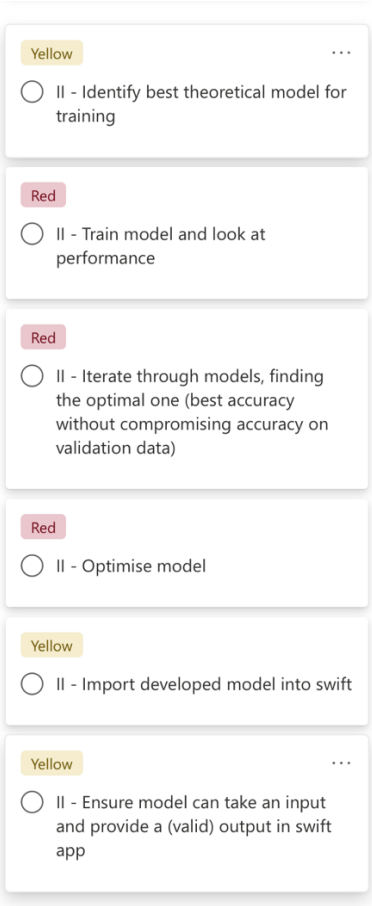
In this sprint I have managed to collect the data to develop the predictive features of my app. I have made sure that the data is appropriate for the ML model, that the labels are correct for the model and cleaned all the data so that it leads to an accurate ML model. The nature of these tickets mean that I was testing the quality of the data as I carried out the tasks in the tickets

This sprint has worked on the following points specified in the design section:

- [1.a.1] Skin Classifier
- [1.a.2] Certainty of Prediction

3. Sprint 2 – Incorporation of ML model into app

a. Ticket overview



b. Ticket 1 – Identify best theoretical model for training

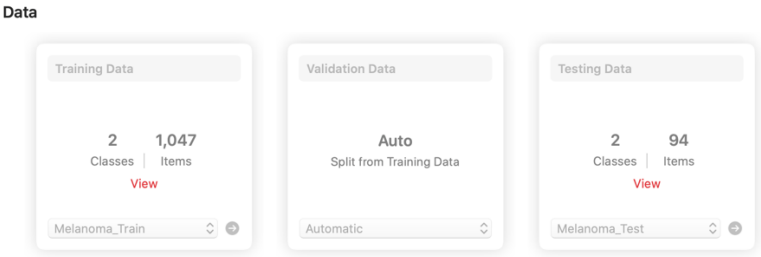
As shown in the design section, I identified that a classification machine learning model (implemented using convolutional neural networks) would be the best for the problem of classifying melanoma. Fortunately, swift has its own "Create ML" application which massively assists in training those models. Therefore, I downloaded create ML and learnt how to use it in order to train a classification model using the data I collected in the sprint before.

c. Ticket 2 - Train model and look at performance

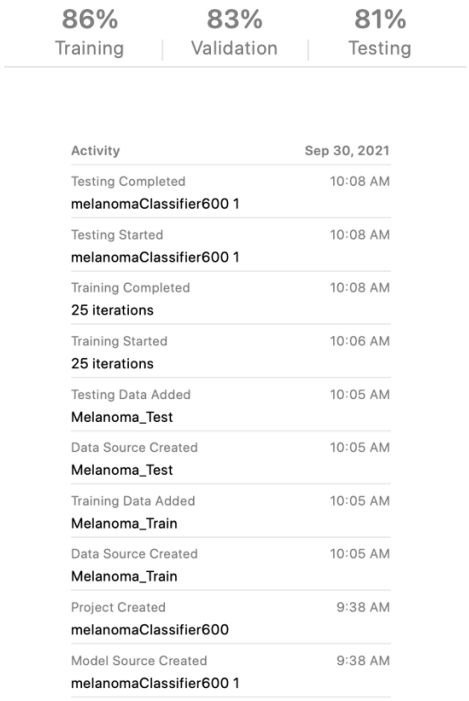
Using create ML I created a new classification model that takes in an image and returns the classification of that image as well as the certainty of the prediction. First, I checked the training worked as I had previously perceived by training a model on a small dataset (only 50 images for each class). This resulted in a model that had around a 60% accuracy, which is essentially just guessing. The

training accuracy was relatively high, but the validation and test accuracy weren't implying there was a lack of data for the model to find correlations between the data and classes.

After doing some trial training, I trained a model with all the data I had (just the raw, unmanipulated data). This resulted in the model training on the data summarised below.



Below is the summary of the model after training for 25 iterations. Training took just under 3 minutes with the raw data, and the model performed relatively well for the first attempt. However, the 81% accuracy for the test data simply wasn't good enough for a melanoma classifier, as that implies around 1 in 5 images would be classified wrong. The fact that the training accuracy was just 86% showed that the model was not overfitting to the data, and that significant improvement would be possible with more data.



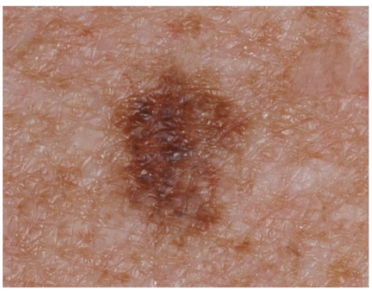
d. Ticket 3 - Iterate through models, finding the optimal one

I realised after training the first melanoma classifier model that more data is needed to create a more accurate model. Therefore, I decided to rotate all the images and therefore increase my data 4-fold. This works as the orientation of the image doesn't affect whether an image contains melanoma or not, and these images, to a computer, are completely new. Therefore, the computer focuses more on learning what patterns (in whatever rotation) makes a mole cancerous rather than learning via brute force. Fortunately Create ML has an inbuilt feature that rotates all the images and then trains on that data, and so I used that feature. The training and test data remained the same as the previous iteration. The results of the training are summarised below.

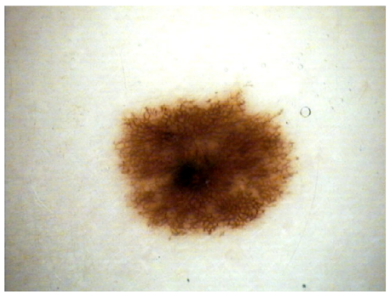
97%	85%	91%
Training	Validation	Testing

Testing Completed melanomaClassifier600 2	10:24 AM
Testing Started melanomaClassifier600 2	10:24 AM
Training Completed 25 iterations	10:24 AM
Training Started 25 iterations	10:12 AM
Testing Data Added Melanoma_Test	10:12 AM
Training Data Added Melanoma_Train	10:12 AM
Model Source Created melanomaClassifier600 2	10:12 AM
Data Source Created Melanoma_Test	10:05 AM
Data Source Created Melanoma_Train	10:05 AM
Project Created melanomaClassifier600	9:38 AM

This model performed significantly better, achieving 91% accuracy on the training data. It also took significantly longer to train (12 minutes) due to the sheer volume of the data being used. Although this model is not good enough for a final application, I noticed something whilst manually testing some data. Some of the images that were classified wrong had significantly lower certainty values than the ones that were classified correctly, therefore I could use the certainty values to let the user know how likely it is that a certain prediction is valid. Below I've shown two cases – the left one is a correctly classified image, the right one is a false positive – and the images show the certainties of those predictions. I will probably make the model better later on if I have more time, but for now this model is sufficient.



Malignant
99% confidence



Malignant
83% confidence

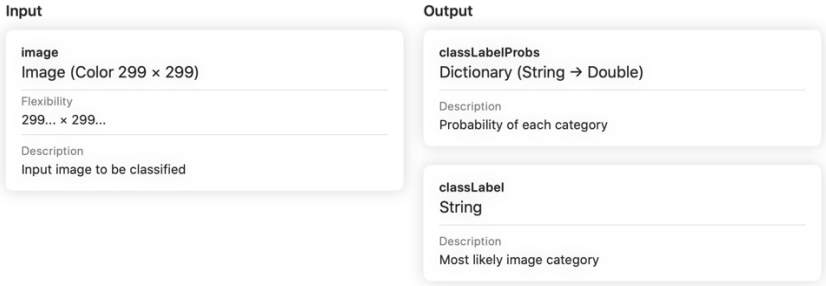
Benign
16% confidence

e. Ticket 4 – Optimise model

The model was optimised through iteration (ticket 3), I will attempt to optimise the model more later on if I have time (as the current model is good enough).

f. Ticket 5 – Import developed model into swift

Having created the melanoma detection model using create ML, I exported it as a ".mlmodel" file from createML. This file lets me use the model as a class inside swift. Below are the inputs and outputs I specified for the model earlier.



My first attempt to use the model is shown below.

```
//
//  melanomaModel.swift
//  MelanomaScan
//
//  Created by Yadav, Aasmaan (SPH) on 30/09/2021.
//

import Foundation

import Vision
import CoreML


let model = try VNCoreMLModel(for: melanomaModel1_91_().model)  2  Call can throw, but errors cannot be thrown out of a global variable ini...
let request = VNCoreMLRequest(model: model, completionHandler: myResultsMethod)
let handler = VNImageRequestHandler(url: URL(fileURLWithPath: "ISIC_1135213.jpg"))
handler.perform([request])                                                  2  Call can throw but is not marked with 'try'

func myResultsMethod(request: VNRequest, error: Error?) {
    guard let results = request.results as? [VNClassificationObservation]
        else { fatalError("huh") }
    for classification in results {
        print(classification.identifier, // the scene label
              classification.confidence)
    }
}

```

This attempt failed because I was attempting to input images into the model which had a .jpg format. Instead, I had to convert the image into a "CVPixelBuffer" format, which is essentially a format that contains a matrix of numbers representing images. The melanoma classifier I made had to necessarily take that as an input, so I had to make subroutines that converted normal images to CVPixelBuffers. The code for this is shown below.

```swift
import Foundation
import UIKit

extension UIImage {

    func resizeTo(size: CGSize) -> UIImage? { //resizes an image to the specified size
        UIGraphicsBeginImageContextWithOptions(size, false, 0.0) // configures the drawing environment for rendering into a bitmap
        self.draw(in: CGRect(origin: CGPoint.zero, size: size)) //redraws the image, scaling it to fit the specified size
        let resizedImage = UIGraphicsGetImageFromCurrentImageContext()! //returns the image from the bitmap context
        UIGraphicsEndImageContext() //clears the rendering work stack
        return resizedImage //returns resized image
    }

    func toBuffer() -> CVPixelBuffer? { //returns the CVPixelBuffer format of an image. Converting a UIImage to a CVPixelbuffer
        let attrs = [kCVPixelBufferCGImageCompatibilityKey: kCFBooleanTrue,
            kCVPixelBufferCGBitmapContextCompatibilityKey: kCFBooleanTrue] as CFDictionary
        var pixelBuffer : CVPixelBuffer?
        let status = CVPixelBufferCreate(kCFAllocatorDefault, Int(self.size.width), Int(self.size.height),
            kCVPixelFormatType_32ARGB, attrs, &pixelBuffer) //alter the kCVPixelFormat type to ensure model is classifying optimally

        guard (status == kCVReturnSuccess) else {
            return nil
        }
        CVPixelBufferLockBaseAddress(pixelBuffer!, CVPixelBufferLockFlags(rawValue: 0))
        let pixelData = CVPixelBufferGetBaseAddress(pixelBuffer!)

        let rgbColorSpace = CGColorSpaceCreateDeviceRGB()

        let context = CGContext(data:pixelData, width: Int(self.size.width), height:
            Int(self.size.height), bitsPerComponent: 8, bytesPerRow:
            CVPixelBufferGetBytesPerRow(pixelBuffer!), space: rgbColorSpace, bitmapInfo:
            CGImageAlphaInfo.noneSkipFirst.rawValue)

        context?.translateBy(x: 0, y: self.size.height)
        context?.scaleBy(x: 1.0, y: -1.0)

        UIGraphicsPushContext(context!)
        self.draw(in: CGRect(x: 0, y: 0, width: self.size.width, height: self.size.height))
        UIGraphicsPopContext()
        CVPixelBufferUnlockBaseAddress(pixelBuffer!, CVPixelBufferLockFlags(rawValue: 0))

        return pixelBuffer
    }
}

```

After creating those functions, I created a very basic view page which could display melanoma images and call a subroutine when a button is pressed. That subroutine would be responsible for using the melanoma model and returning a classification and certainty of classification for a certain image. This subroutine would make use of the CVPixelBuffer converter function mentioned above to ensure that the input to the melanoma model is correct.

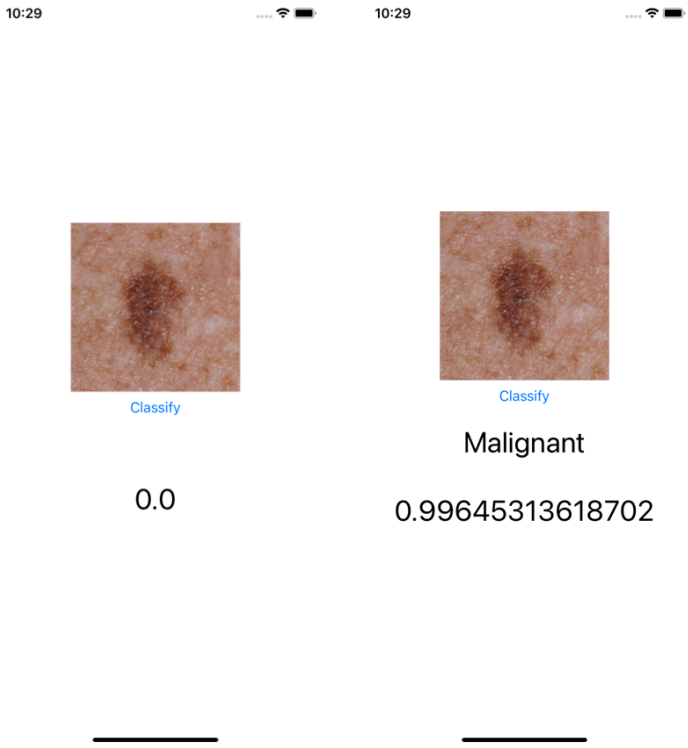
Code for view model is shown below.

```swift
import SwiftUI

struct ContentView: View {

    //display variables to do with the output of image classification
    @State private var currentImageName = "testImageMalignant1"
    @State private var classificationLabel: String = ""
    @State private var confidence: Double = 0

    //creates instance of imageClassifier class
    let imageClassifierInstance = imageClassifier()

    var body: some View {
        VStack {
            //displaying chosen image, as well as information about classification of that image
            Image(currentImageName)
                .resizable()
//                .frame(width:UIScreen.main.bounds.width*(3/4), height:UIScreen.main.bounds.height*(1/4))
                .frame(width:200,height:200)

            //button to call subroutine to classify image
            Button("Classify") {
                (self.classificationLabel, self.confidence) =
                    imageClassifierInstance.performImageClassification(imageName: currentImageName)
            }

            Text(classificationLabel)
                .padding()
                .font(.largeTitle)

            Text(String(confidence))
                .padding()
                .font(.largeTitle)
        }
    }

}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

In this piece of code, I create a view to show something on an iPhone screen. I minimise using any logic in this file, as I do not want that logic to interfere with the user interface code. Therefore, I put the logic for the subroutine that classifies the image in a class called imageClassifier (line 18), and called a subroutine that that class contains on line 30 instead of creating the subroutine in this file. The code for the imageClassifier class is shown below.

```swift
import Foundation
import SwiftUI

class imageClassifier {

    //creating an instance of the trained melanoma model.
    let model = melanomaModel1_91_()  ⚠ 'init()' is deprecated: Use init(configuration:) instead and handle errors appropriately.

    //performs classification on the image specified, returning the output classification as well as the
        certainty of the classification.
    func performImageClassification(imageName: String) -> (String, Double) {

        guard let img = UIImage(named: imageName),
              let resizedImage = img.resizeTo(size: CGSize(width:600, height:600)),
              let buffer = resizedImage.toBuffer() else {
            return ("error1", 101.1)
        }

        let outputOptional = try? model.prediction(image: buffer)

        if let output = outputOptional {

            if let malignantProbability = output.classLabelProbs["Malignant"] {
                return (output.classLabel, malignantProbability)
            }
            return (output.classLabel, 101.3)
        }

        return ("error2", 101.2)
    }

}
```

In this class I first import the machine learning model I had created (which I had name MelanomaModel1_91_ - referring to its accuracy in the testing phase). Then I convert the image sent in from the view into the CVPixelBuffer format. If that has been done successfully the subroutine will make a prediction using the melanoma model and return a prediction along with the certainty of that prediction. Below I have shown the state of the app, from a user's point of view, after this ticket.



The app has a very basic but functional look. You have an image, and you can press classify to classify it, as well as return the certainty of that prediction.

g. Ticket 6 – Ensure model can take an input and provide a (valid) output in swift app

To test that everything is working correctly, I manually inputted the images used by the app back into the create ML model, to ensure that the model was still making the same predictions with the same certainty as before. This also helped me make sure that I had not got the CVPixelBuffer converter wrong (ie the values for 'red' were not swapped with thee values for 'alpha' in the image representation). I checked the values, and everything worked as expected.

h. Sprint Review

In this sprint I have created the machine learning model that is the main feature of the app. This sprint was mostly iterative, and so involved me finding ways to improve the current model rather than testing it to find errors. I did test the UI and the interaction of the model with the user, which seems to work at this point. I will continue testing what I have created so far after each sprint to make sure nothing new has affected the current development.

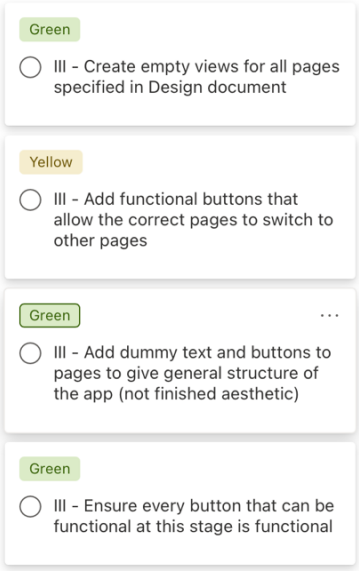
At this point I also decided to get some feedback from Dr Lyman, who expressed his concern for the accuracy of the initial model (81% testing accuracy). This is what caused me to further improve the model, so that it has more than 90% accuracy. He then mentioned that 91% testing accuracy was very good for an initial app, but if the app was to go into production it would have to have at least 99% accuracy due to the sensitivity of the topic.

This sprint has worked on the following points specified in the design section:

- [1.a.1] Skin Classifier
- [1.a.2] Certainty of Prediction

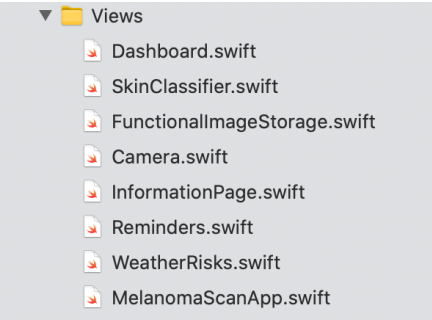
4. Sprint 3 – Views Creation

a. Ticket overview



b. Ticket 1 – Create empty views for all pages specified in Design document

I created new view files in swift, with each file having a name corresponding to the view it specified from the design document. All the files I created for this ticket are shown below.



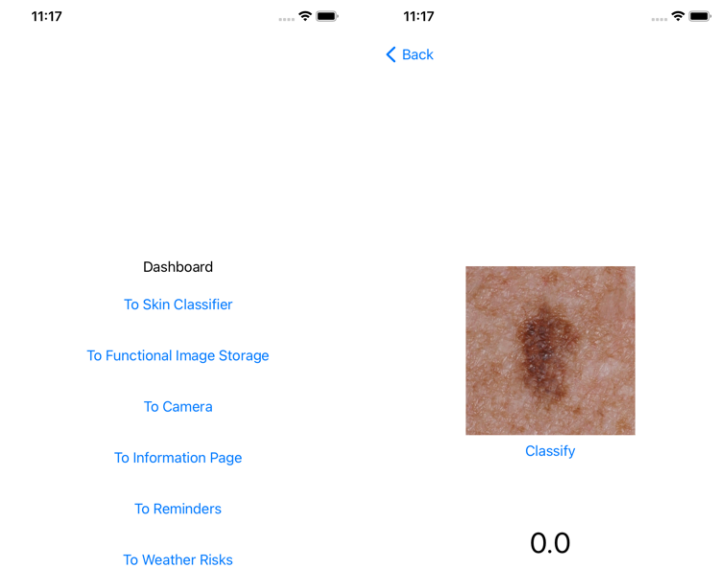
I kept each view to a minimum, only adding some text that helped me identify what view it was, so that I could check that whilst testing. Also, it made no sense to add more UI with no functional code developed for that view. The code and look of each of the views is shown below.

```
8  import SwiftUI
9
10 struct WeatherRisks: View {
11     var body: some View {
12         Text("Weather Risks Page")
13     }
14 }
15
16 struct WeatherRisks_Previews: PreviewProvider {
17     static var previews: some View {
18         WeatherRisks()
19     }
20 }
21
```

c. Ticket 2 – Add functional buttons that allow the correct pages to switch to other pages

In order to create functional buttons that can switch between pages, I decided to use navigation views. This type of view allows me to go to different pages, and then have an inbuilt ‘back’ button that leads back to the previous page automatically. I decided to make my dashboard the ‘root’ navigation view, meaning that if you press the back button continuously, it will lead you to the dashboard. The code and look of the dashboard view is shown below.

```swift

import SwiftUI

struct Dashboard: View {
    var body: some View {

        // a navigation view allows for links to other pages
        NavigationView {
            VStack {
                Text("Dashboard")

                //links to other pages.
                NavigationLink("To Skin Classifier", destination: SkinClassifier())
                    .padding()

                NavigationLink("To Functional Image Storage", destination:
                    FunctionalImageStorage())
                    .padding()

                NavigationLink("To Camera", destination: Camera())
                    .padding()

                NavigationLink("To Information Page", destination: InformationPage())
                    .padding()

                NavigationLink("To Reminders", destination: Reminders())
                    .padding()

                NavigationLink("To Weather Risks", destination: WeatherRisks())
                    .padding()
            }
        }

    }
}

struct Dashboard_Previews: PreviewProvider {
    static var previews: some View {
        Dashboard()
    }
}

```



Now the app has a dashboard page that can lead to any other page and then lead back to the dashboard with ease. I tested this by using the app, opening all the pages and making sure that I could return to the dashboard from all the pages.

d. Ticket 3 – Add dummy text and buttons to pages to give general structure of the app

I used navigation link buttons to create pathways between pages, making sure that the app could traverse across pages as required. The code for each of the links followed the form shown below.

```
NavigationLink("To Camera (choose new img)", destination: Camera())
```

e. Ticket 4 – Ensure that every button that can be functional at this stage is functional

For this ticket I tested the buttons I made previously by clicking on them and making sure every button worked and returned me to the dashboard with the back buttons. I also went through my design document to check that every button that can currently have a functional purpose existed.

f. Sprint Review

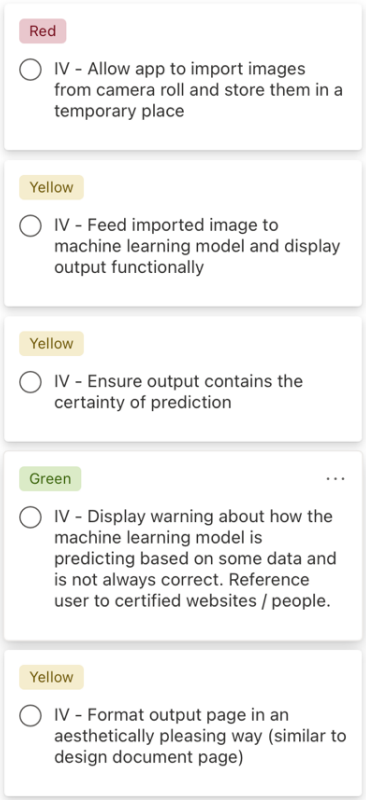
This sprint was not very algorithmically complex, and so did not require many tests. I ensured that all the buttons I created worked, and let five different people use the app in an attempt to find any bugs, and fortunately none were found.

This sprint has worked on the following points specified in the design section:

- General structure of app

5. Sprint 4 - Images

a. Ticket overview



b. Ticket 1 – Allow app to import images from camera roll and store them in a temporary place

In order to make sure my app works on images of skin taken from iPhone, I decided to allow users to import already taken images from their camera roll into the app.

```
            //displaying chosen image, as well as information about classification of that image
            Button(action: {
                changeClassificationImage = true
                openCameraRoll = true

            }, label: {
                if changeClassificationImage == true {
                    Image(uiImage: imageSelectedFromCameraRoll)
                        .resizable()
                        .frame(width: 250, height: 250)
                } else {
                    Image(initialImageName)
                        .resizable()
                        //.frame(width:UIScreen.main.bounds.width*(3/4), height:UIScreen.main.bounds.height*(1/4))
                        .frame(width:250,height:250)
                }
            })

```

The code above creates a button that displays an image on the skin classifier view. The image represents what image the user has currently selected to classify, and if the image is clicked the camera roll is opened (which is done by the code shown below).

```
        }.sheet(isPresented: $openCameraRoll, content: {
            ImagePicker(selectedImage: $imageSelectedFromCameraRoll,sourceType: .photoLibrary)
        })

```

This calls the class “ImagePicker”, which contains the code to open a sheet that shows all the images the user can select from. I have shown the code in “ImagePicker” below.

```
import Foundation
import SwiftUI
import UIKit


struct ImagePicker: UIViewControllerRepresentable {

    @Binding var selectedImage: UIImage
    @Environment(\.presentationMode) private var presentationMode

    //defines the source of the UIImagePickerControllere
    var sourceType: UIImagePickerController.SourceType = .photoLibrary

    //Makes the controller, allowing the app to interact with the photo library
    func makeUIViewController(context: UIViewControllerRepresentableContext<ImagePicker>) -> UIImagePickerController {

            let imagePicker = UIImagePickerController()
            imagePicker.allowsEditing = false
            imagePicker.sourceType = sourceType
            imagePicker.delegate = context.coordinator

            return imagePicker
    }

    func updateUIViewController(_ uiViewController: UIViewControllerType, context: Context) {
        // add stuff later if needed
    }

    //Actual picking of the image
    final class Coordinator: NSObject, UIImagePickerControllerDelegate, UINavigationControllerDelegate {

        var parent: ImagePicker
        init(_ parent: ImagePicker) {
            self.parent = parent
        }

        //makes sure image is returned correctly after it is picked. Saving the image after it is picked.
        func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

            if let image = info[UIImagePickerController.InfoKey.originalImage] as? UIImage {
                parent.selectedImage = image
            }

            parent.presentationMode.wrappedValue.dismiss()
        }
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }
}

```

c. Ticket 2 – Feed imported image to machine learning model and display output functionally

This ticket was relatively simple having done the ticket before. I just changed the image selected in the skin classifier view to the one chosen by the user. I also had to make sure the app updated the image

sent to the skin classifier every time it was changed, and therefore had to make the image into a state object.

```
20      //picture picker
21      @State var changeClassificationImage = false
22      @State var openCameraRoll = false
23      @State var imageSelectedFromCameraRoll = UIImage()
51  Button("Classify") {
52      showingClassificationWarning = true
53      (self.classificationLabel, self.confidence) =
            imageClassifierInstance.performImageClassification2(image: imageSelectedFromCameraRoll)
54  }
```

d. Ticket 3 – Ensure output contains the certainty of prediction

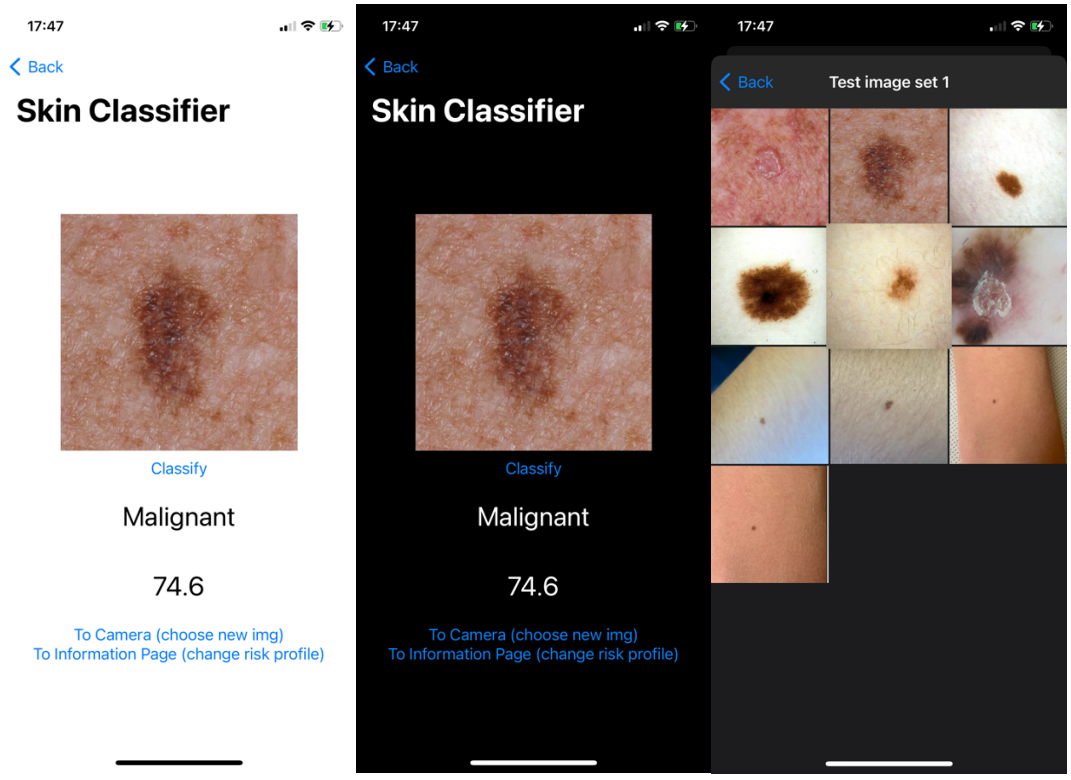
For this ticker I had to design a function that would take in the raw, unadjusted certainty value from the melanoma classifier model and convert it into a normalised, usable percentage value. The output of the melanoma classification model is the classification of the image, alongside a value between 0 and 1.

A value of 0 would represent 100% certainty in the image containing a benign mole, and a 1 would represent a 100% certainty in the image containing melanoma. This value requires a lot of explaining and is not very usable/intuitive, so I created the function shown below to convert from this value into a percentage value between 0 and 100, that accompanies the classification of the image. This new value shows how certain the prediction is on that particular class (ie 34% certain about melanoma being present rather than a value of 6.7).

The code for this function is shown below.

```
62  //takes the raw certainty value from the melanoma ml model, and converts it into a usable certainty
63  func certaintyFunction(oldCertainty: Double) -> Double {
64      //function to take in pure probability of prediction and return a certainty (ie measure from 50% rather than from 0%
65      var workingCertainty: Double = oldCertainty - 0.5
66      workingCertainty = workingCertainty * 2
67      workingCertainty = sqrt(workingCertainty * workingCertainty)
68      workingCertainty = workingCertainty * 100
69
70      return workingCertainty
71  }
```

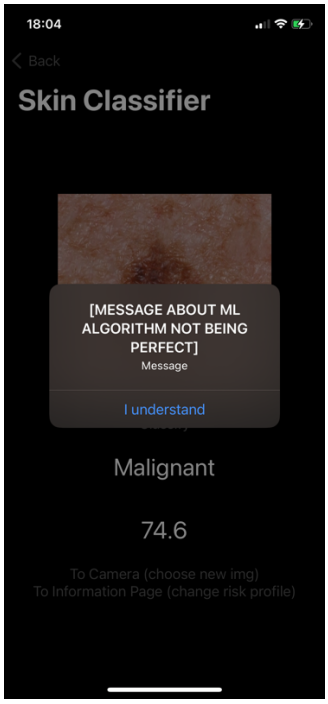
The screenshots below show the UI after tickets 1,2 and 3. I have also made sure that my app is compatible with apple's dark mode, and the UI in dark mode is also shown below. The third image shows the image selector sheet, allowing the user to choose another image for classification.



e. Ticket 4 – Display warning about how the machine learning model is predicting based on some data and is not always correct. Reference user to certified websites / people.

To implement this ticket, I decided to use iOS alerts. This would allow me to create an alert that the user must read before going on to see the result of the classification, ensuring that the user is aware of the shortcomings of this implementation. I decided that it would be ineffective and unusable to put the links to certified websites in the alert, and instead will put them in the information page.

```
57              .alert( isPresented: $showingClassificationWarning) {
58                  Alert(
59                      title: Text("[MESSAGE ABOUT ML ALGORITHM NOT BEING PERFECT]"),
60                      message: Text("Message"),
61                      dismissButton: .default(Text("I understand"), action: {
62                      })
63                  )
64              }
```



f. Ticket 5 – Format output page in an aesthetically pleasing way (similar to design document page)

I decided that this would be best done after all the functional parts of the app are developed. This allows me to functionally change something majorly down the line without having to re-do the aesthetics of the app due to that change.

h. Sprint Review

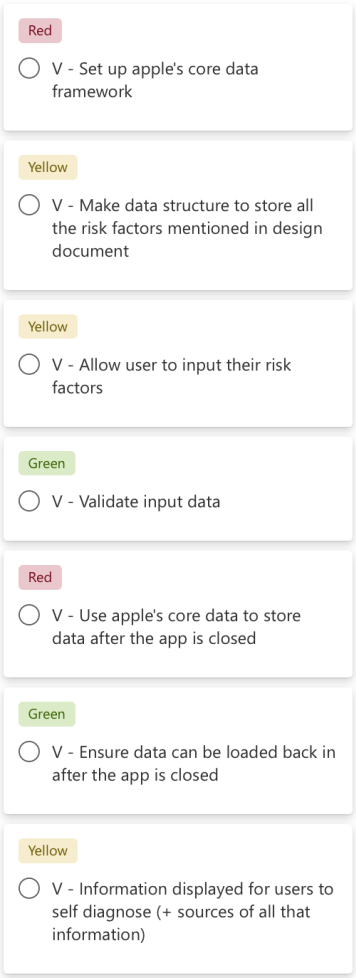
In this sprint I focussed on ensuring that the skin classifier page has all the functionality that was mentioned in the design document. I will make the page (and the rest of the app) more aesthetically pleasing once the functionality of the app is finished. To further test the sprint as a whole (having tested each ticket individually in the ticket section), I gave the app to 5 people to attempt to break. There were no major errors, but there was one problem that no image would show if the user exited the image selector without clicking on an image – something I will fix in the aesthetics sprint.

This sprint has worked on the following points specified in the design section:

- [1.a.1] Skin Classifier
- [1.a.3] Functional Image Storage

6. Sprint 5 – Risk Factors and Database

a. Ticket overview

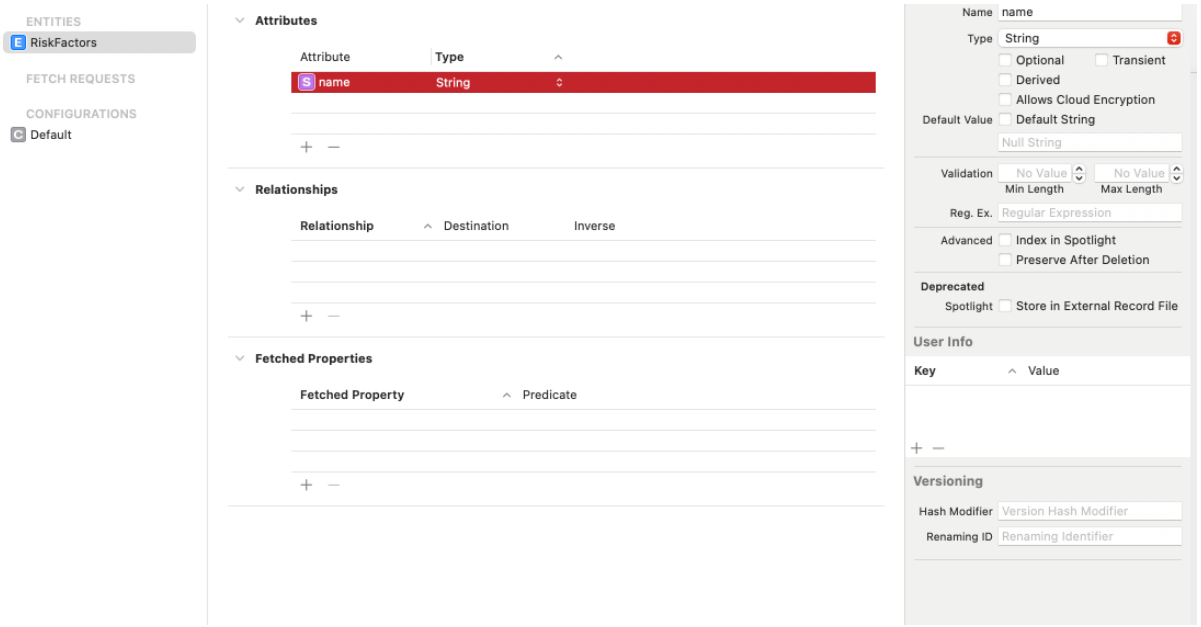


b. Ticket 1a – Set up apple's core data framework

I decided to use apple's Core Data framework to have data persistence on my app. The framework is quite large, and requires a lot of setting up, but is very compatible with swiftUI and is scalable if I want to further improve the app in the future.

To set up Core Data within my app, I decided to create a data structure with only one string being stored. This allowed me to develop my understanding of the framework easily, and it seemed like a good way to integrate Core Data, as I could always add more data in the future with relative ease.

I created a data structure that is represented below. It only contains an attribute called "name", which I used to test how everything works, and whether it works at all.



Then I created a class called “CoreDataManager”, which is responsible for all the requests to fetch, save and delete data from memory. It allows me to access the Core Data model shown above. In the class I created two basic functions – one to save the name attribute, and one to fetch all the saved attributes.

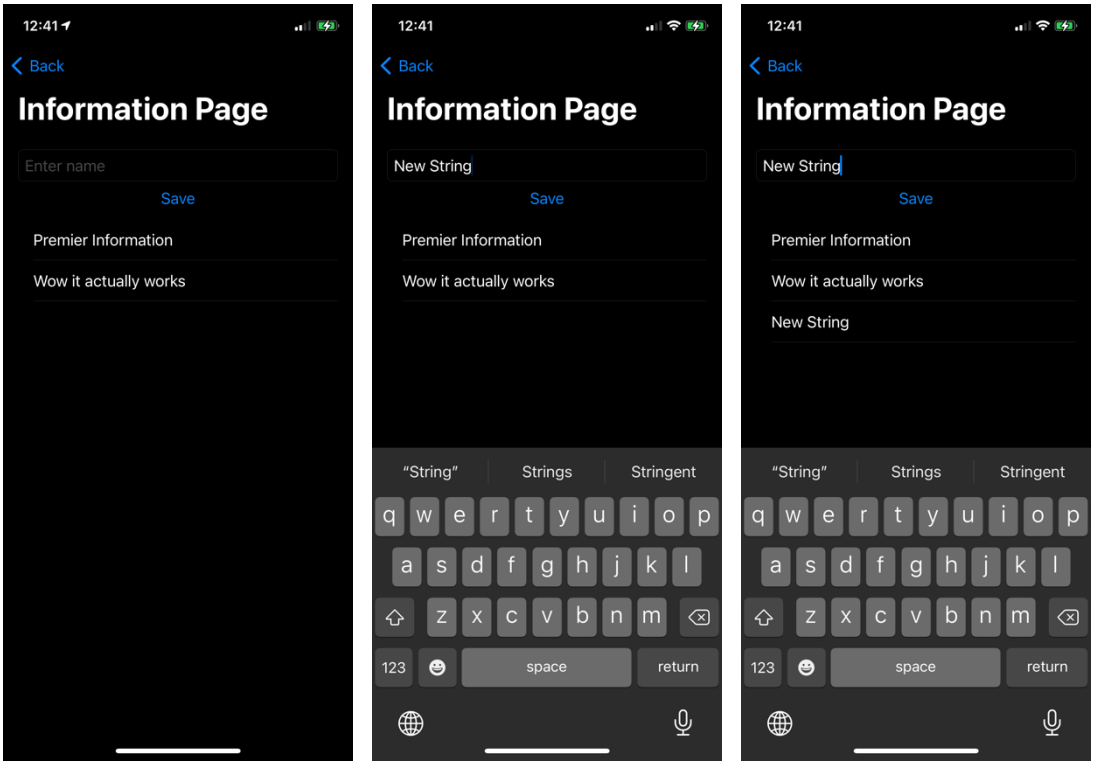
```swift
import Foundation
import CoreData

class CoreDataManager {

    //responsible for loading in the model
    let persistentContainer: NSPersistentContainer

    init() {
        persistentContainer = NSPersistentContainer(name: "RiskFactorsPersistence")
        persistentContainer.loadPersistentStores { (description, error) in
            if let error = error {
                fatalError("Core Data store failed \(error.localizedDescription)")
            }
        }
    }

    //saves the current risk factor (only the name in this case)
    func saveRiskFactorName(name: String) {
        //instance of riskFactors
        let riskFactors = RiskFactors(context: persistentContainer.viewContext)
        riskFactors.name = name

        do {
            try persistentContainer.viewContext.save()
        } catch {
            print("Failed to save risk factors \(error)")
        }
    }

    //returns all the saved ages
    func getAllNames() -> [RiskFactors] {

        let fetchRequest: NSFetchRequest<RiskFactors> = RiskFactors.fetchRequest()

        do {
            return try persistentContainer.viewContext.fetch(fetchRequest)
        } catch {
            return []
        }
    }
}

```

Next, I implemented some basic UI in the InformationPage file, which allowed me to test everything I had currently built. The code is shown below.

```swift
import SwiftUI

struct InformationPage: View {

    let coreDM: CoreDataManager
    @State private var riskFactorName: String = "" //convert to Int16 later
    //use a viewmodel for the variable below again.
    //state is used to make sure user interface is in sync with the data.
    @State private var names: [RiskFactors] = [RiskFactors]()

    //again into a VM model
    private func populateNames() {
        names = coreDM.getAllNames()
    }

    var body: some View {
        VStack {
            //input risk factors and save using the save button
            TextField("Enter name", text: $riskFactorName)
                .textFieldStyle(RoundedBorderTextFieldStyle())
            Button("Save") {
                //create a view model layer, and call this from that class -NOT THE UI
                coreDM.saveRiskFactorName(name: riskFactorName)
                populateNames()
            }

            List(names, id: \.self) { name in
                Text(name.name ?? "")
            }

        }.padding()
        .navigationBarTitle("Information Page")

        .onAppear(perform: {
            populateNames()
        })

    }
}

struct InformationPage_Previews: PreviewProvider {
    static var previews: some View {
        InformationPage(coreDM: CoreDataManager())
    }
}

```

For testing purposes, I did include some logic in the view file. This is highlighted by the comments and will be moved to a different class in a different file later on, after making sure everything works well.

Below I have included a few screenshots of what this code has accomplished. It has allowed me to enter a string and press save. The string in the text box is saved to memory when the save button is pressed and shown in a list below. The saving to memory allows me to close the app, restart my phone, and the data will still be present in the list.

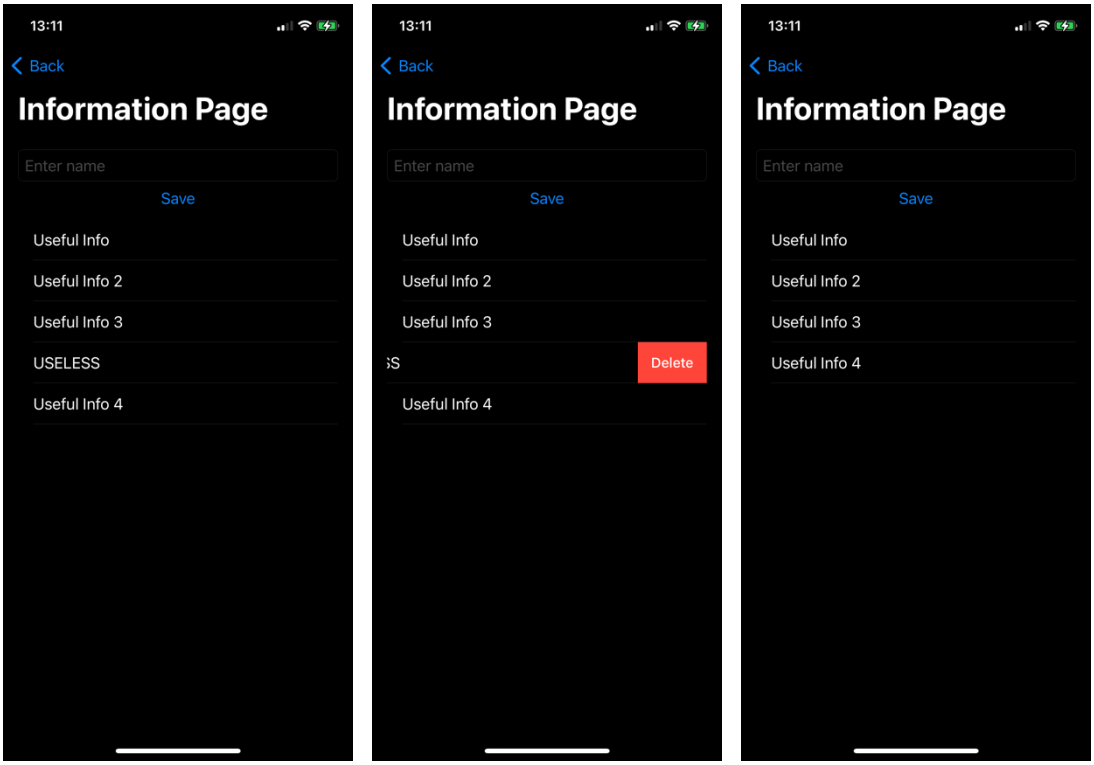


I also had to make sure that data can be deleted, so I implemented a new function in the Core Data manager to do just that. I also made sure to include "do { } catch { }" in all these functions to make sure that there is no case in which the app will crash. If something wrong does happen, the app will display an error but not execute the erroneous code. Also, I made sure in the following code to roll back to the previous working state of data storage if some error does occur.

```
50      //deletes from memory
51      func deleteName(riskFactor: RiskFactors) {
52
53          persistentContainer.viewContext.delete(riskFactor)
54
55          do {
56              try persistentContainer.viewContext.save()
57          } catch {
58              persistentContainer.viewContext.rollback()
59              print("Failed to save context \(error)")
60          }
61      }
62  }
63
```

```
34              //displays names in a way that they can be deleted
35              List {
36                  ForEach(names, id: \.self) { name in
37                      Text(name.name ?? "")
38                  }.onDelete(perform: {indexSet in
39                      indexSet.forEach{ index in
40                          let name = names[index]
41                          //delete the name using core data manager
42                          coreDM.deleteName(riskFactor: name)
43                          populateNames()
44                      }
45                  })
46              }
47
```

Below are some screenshots of what this code looks like in the app.



c. Ticket 1b – Set up apple's core data framework

I decided to split this ticket into two, as it was much more challenging than I had thought it would be previously. I needed to make sure that every risk factor in the app was editable, so I decided to make a page that would open once you click on any risk factor. This page would allow you to update values for that risk factor. The code for the UI page is shown below.

```swift
import SwiftUI

struct RiskFactorDetailTESTVIEW: View {

    let riskFactor: RiskFactors
    @State var riskFactorName: String = ""
    let coreDM: CoreDataManager

    var body: some View {
        VStack {
            TextField(riskFactor.name ?? "", text: $riskFactorName)
                .textFieldStyle(RoundedBorderTextFieldStyle())
                .padding()
            Button("Update") {
                if !riskFactorName.isEmpty {
                    riskFactor.name = riskFactorName
                    coreDM.updateRiskFactor()
                }
            }.padding()

            Text("ADD MORE INFO ABOUT RISK FACTOR")
                .padding()

        }
    }
}

struct RiskFactorDetailTESTVIEW_Previews: PreviewProvider {
    static var previews: some View {
        let riskFactor = RiskFactors()
        RiskFactorDetailTESTVIEW(riskFactor: riskFactor, coreDM: CoreDataManager())
    }
}
```

```
34              //displays names in a way that they can be deleted
35              List {
36                  ForEach(riskFactorNames, id: \.self) { name in
37                      //NavigationLink(destination: RiskFactorDetailTESTVIEW(riskFactor: name, coreDM: coreDM), label: Text(name.name ?? ""))
38
39                      NavigationLink(destination: RiskFactorDetailTESTVIEW(riskFactor: name, coreDM: coreDM), label: {
40                          Text(name.name ?? "")
41                      })
42
43                  }.onDelete(perform: {indexSet in
44                      indexSet.forEach{ index in
45                          let name = riskFactorNames[index]
46                          //delete the name using core data manager
47                          coreDM.deleteName(riskFactor: name)
48                          populateNames()
49                      }
50                  })
51              }
```

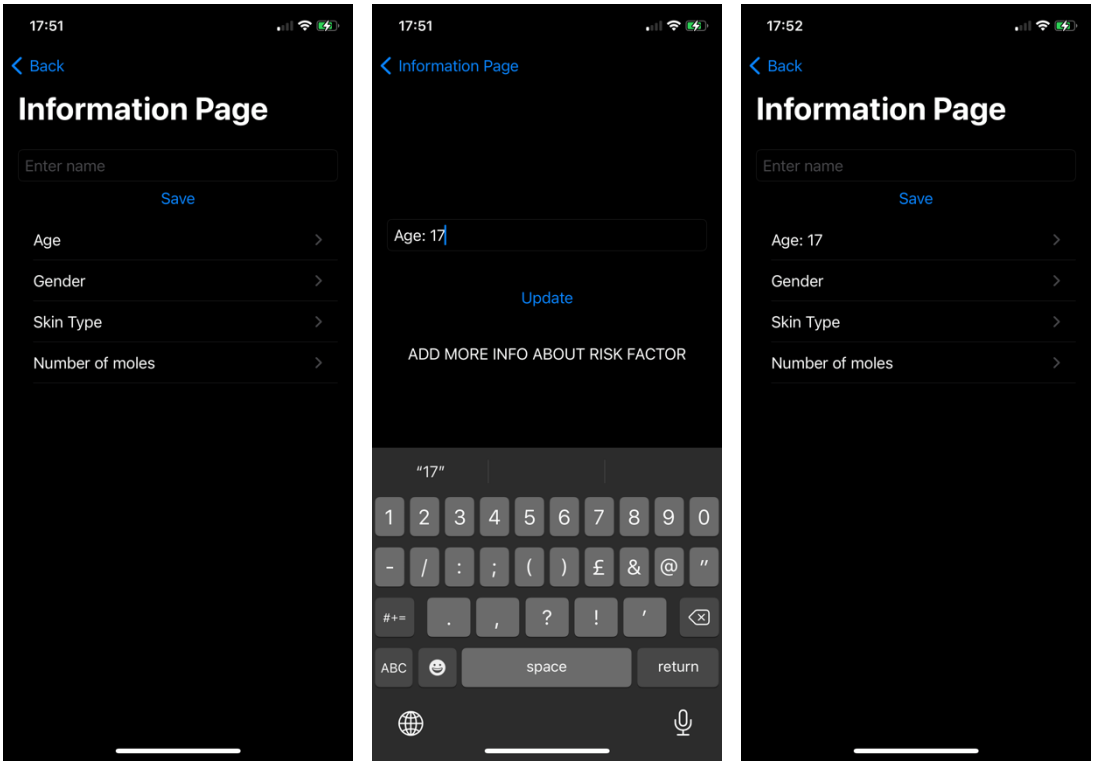
I also added code that enables the risk factors to be updated, rather than just being able to create new ones and delete old ones.

```
63      func updateRiskFactor() {
64
65          do {
66              try persistentContainer.viewContext.save()
67          } catch {
68              persistentContainer.viewContext.rollback()
69          }
70      }
71  }
72
```

In my testing I found a slight error with the framework – it would not properly update the values saved via core data when a page was reloaded. In order to fix this, I made a state variable that changes value whenever data is updated – causing the view to refresh the data and not to used pre-stored values. The code for this is shown below.

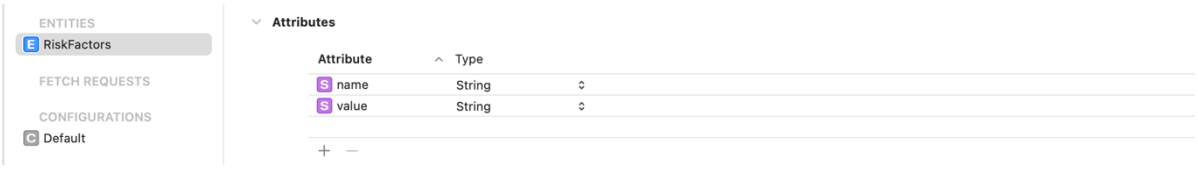
```
17      @State private var needsRefresh: Bool = false
52          }.listStyle(PlainListStyle())
53          .accentColor(needsRefresh ? .black: .white)
22           Button("Update") {
23               if !riskFactorName.isEmpty {
24                   riskFactor.name = riskFactorName
25                   coreDM.updateRiskFactor()
26                   needsRefresh.toggle()
27               }
```

The following images show what this change looks like in the app.



d. Ticket 2 – Make data structure to store all the risk factors mentioned in design document

In the app's information page, I had to display risk factor names, along with the values which the user has input into the app. In order to do this, I had to restructure how I was going to store data. I changed my risk factor storage entity in my core data model to include two attributes to store – the "name" and the "value".



The name stores the name of the risk factor currently being stored, and the value stores the actual data related to the risk factor. As the value is a string type, I will need to validate the input in the future to ensure it can be converted between string and the desired type without any errors.

As the data model was changed, I also had to edit all my read, write, delete and update subroutines. Furthermore, I had to completely restructure my view to accommodate this new data structure, and also to make it significantly easier to implement the next ticket (the code for which is included in the next ticket, as it overlapped with this one).

e. Ticket 3 – Allow user to input their risk factors

A major problem with my implementation of core data before was that any amount of risk factors could be added by the user. This is not what would happen in the final app, as there will be a select set of risk factors to enter, that can be used by different algorithms. In order to solve this I removed the option to add any risk factors, and instead populated the risk factors automatically when the app starts.

To do this I created risk factor entities for every item in the following list when the view.

```
18      let stringListOfRiskFactors = ["Age", "Gender", "Skin Type", "Number of Moles"]
```

Then I called a function to create those risk factors when the app started.

```
21      private func populateRiskFactors() {
22          riskFactorsList = coreDataManager.getAllNames()
23      }
24

53              .onAppear(perform: {
54                  populateRiskFactors()
55              })

40  func getAllNames() -> [RiskFactors] {
41
42      let fetchRequest: NSFetchRequest<RiskFactors> = RiskFactors.fetchRequest()
43
44      do {
45          return try persistentContainer.viewContext.fetch(fetchRequest)
46      } catch {
47          return []
48      }
49  }
```

Once those risk factors are retrieved, I displayed them in a list, with the risk factor's name as the main label, and the value of that risk factor displayed as well.

```
30              ForEach(riskFactorsList, id: \.self) { riskFactor in
31                  NavigationLink(destination: RiskFactorDetail(riskFactor: riskFactor, coreDM: coreDataManager, needsRefresh:
                        $needsRefresh), label: {
32                      HStack {
33                          Text(riskFactor.name ?? "")
34                          Spacer()
35                          Text(riskFactor.value ?? "")
36                      }
37                  })
38              }
39
40          //This is just to make sure that everything stays up to date.
41          }
42          .listStyle(PlainListStyle())
43          .accentColor(needsRefresh ? .black: .white)
```

Every item in the list is also clickable, leading to another view that allows you to edit the value of that risk factor. The code for that view is shown below.

```swift
import SwiftUI

struct RiskFactorDetail: View {

    let riskFactor: RiskFactors
    @State var riskFactorName: String = ""
    let coreDM: CoreDataManager
    @Binding var needsRefresh:Bool

    var body: some View {
        VStack {

            Text(riskFactor.value ?? "")

            TextField(riskFactor.value ?? "", text: $riskFactorName)
                .textFieldStyle(RoundedBorderTextFieldStyle())
                .padding()
            Button("Update") {
                if !riskFactorName.isEmpty {
                    riskFactor.value = riskFactorName
                    coreDM.updateRiskFactor()
                    needsRefresh.toggle()
                }
            }.padding()

            Text("ADD MORE INFO ABOUT RISK FACTOR")
                .padding()

        }.navigationBarTitle(riskFactor.name ?? "")
    }
}

struct RiskFactorDetailTESTVIEW_Previews: PreviewProvider {
    static var previews: some View {
        let riskFactor = RiskFactors()
        RiskFactorDetail(riskFactor: riskFactor, coreDM: CoreDataManager(), needsRefresh: .constant(false))
    }
}

```

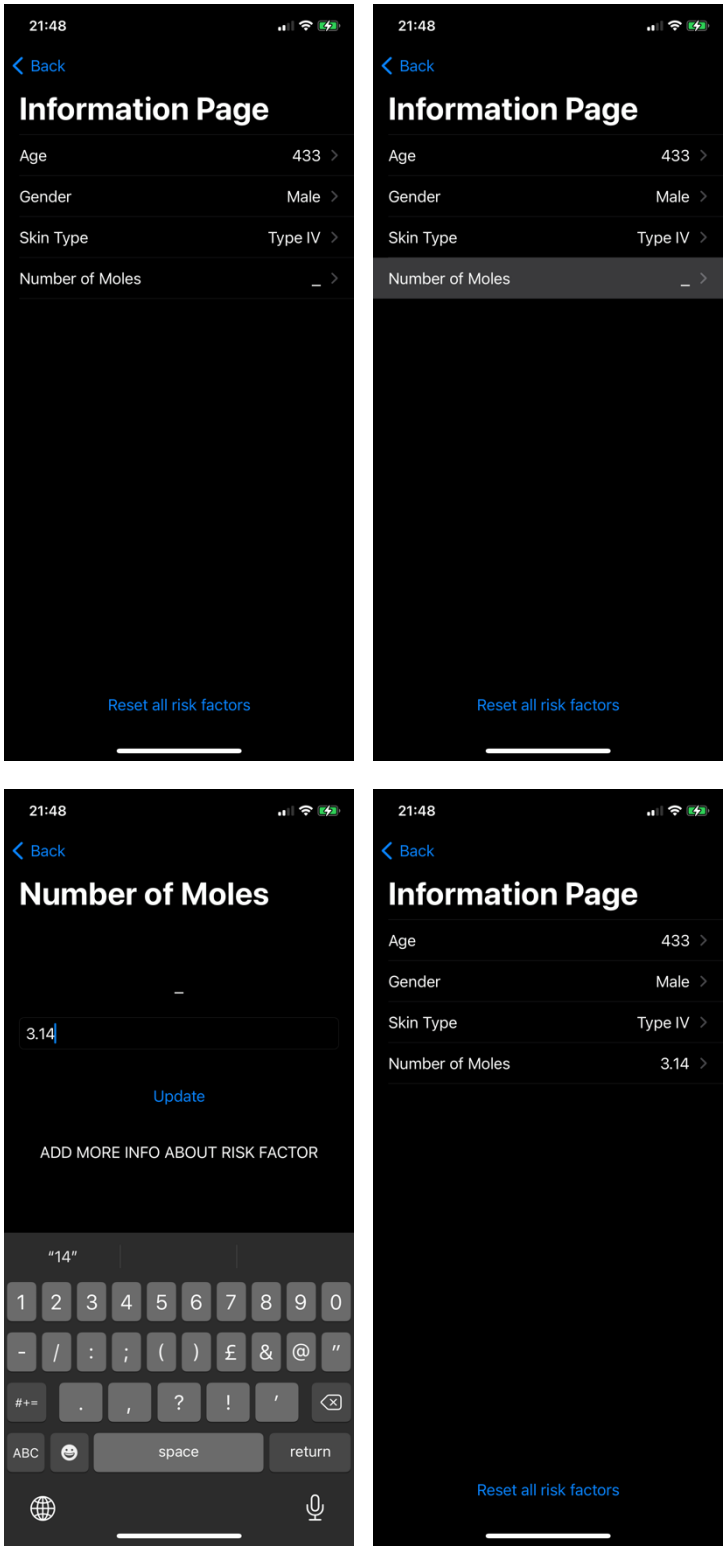
In the information page I decided to add a “reset risk factors” button that would allow users to completely reset all their risk factors in case something goes wrong. The button works by deleting all the information in the data store, then recreating that data with empty values for all the risk factors given in the list above.

```swift
        Button("Reset all risk factors") {
            // coreDataManager.saveRiskFactorValue(stringValue: riskFactorValue)
            coreDataModel.resetRiskFactors(coreDataManager: coreDataManager, listOfRiskFactors: stringListOfRiskFactors)
            populateRiskFactors()
            //needsRefresh.toggle()
        }.padding()
```

```swift
    func resetRiskFactors(coreDataManager: CoreDataManager, listOfRiskFactors: [String]) {

        coreDataManager.deleteAllRiskFactors()

        for item in listOfRiskFactors {
            coreDataManager.saveRiskFactorValue(stringValue: item)
        }

    }
```

```swift
    func deleteAllRiskFactors() {

        let allRiskFactors: [RiskFactors] = getAllNames()

        for savedRiskFactor in allRiskFactors {
            persistentContainer.viewContext.delete(savedRiskFactor)
        }

        do {
            try persistentContainer.viewContext.save()
        } catch {
            persistentContainer.viewContext.rollback()
            print("Failed to save context \(error)")
        }

    }
```

All this creates the screenshots below, showing how the app is now able to take in a set number of inputs, store the data in memory and reuse it when the app is reopened.



f. Ticket 3 – validate data

In order to make sure all the data was correct for each risk factor, I decided to implement "pickers" which allow the user to select from a set of values from their input, rather than being able to enter anything.

I also realised at this point that I would need to add a new attribute to my data model called “numericalRiskValue”, which is a value between 0 and 5. This value is used to calculate how much risk a certain factor gives (with 0 being no extra risk, and 5 being high additional risk).

To make sure the correct type of data was inputted for every risk factor, I created a function that returns possible options for every risk factor (code shown below).

```
24      func returnStringListOfRiskFactors(name: String) -> [String] {
25
26          if name == "Age"{
27
28              return ["0-9", "10-19", "20-39", "40-69", "70-99", "100+"]
29
30  //              var output: [String] = []
31  //              for i in 0...130 {
32  //                  output.append(String(i))
33  //              }
34  //              return output
35
36          } else if name == "Gender" {
37
38              return ["Male", "Female", "Other"]
39
40          } else if name == "Skin Type" {
41
42              return ["Type 1", "Type 2", "Type 3", "Type 4", "Type 5", "Type 6"]
43
44          } else if name == "Eye Colour" {
45
46              return ["dark", "light brown", "blonde", "red/red-blonde"]
47
48          } else if name == "Hair Colour" {
49
50              return ["dark", "blue/blue-grey", "green/grey/hazel"]
51
52          } else if name == "Number of Moles" {
53
54              return ["0-20", "20-50", "50-100", "100+"]
55
56          } else if name == "Freckles" {
57
58              return ["Present", "Absent"]
59
60          } else if name == "Family History" {
61
62              return ["None", "1 member", "more than 1 member"]
63
64          } else if name == "Diseases and Conditions" {
65
66              return ["Inflammatory Bowel Disease (IBD)", "Human Immunodeficieny Virus (HIV)", "IBD and HIV", "None"]
67
68          } else if name == "Body Mass Index" {
69
70              return ["<18.5", "18.5-25.9", "26-29.9", "30-34.9", ">35"]
71
72          } else {
73
74              return ["not configured"]
75
```

Then I made the page where you can edit the value of the risk factor modular, so I could just pass the name of the risk factor in, and it would return all the possible values I could choose from. This allows the user to choose any risk factor and update its value, and because there are only certain values the user can choose from, the input is validated.

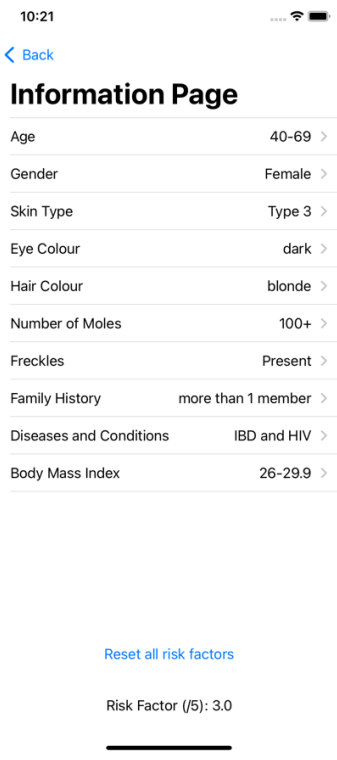
```swift
import SwiftUI

struct RiskFactorDetail: View {

    let riskFactor: RiskFactors
    let coreDM: CoreDataManager
    let riskFactorsModel = RiskFactorsModel()
    @Binding var needsRefresh: Bool
    @State var whichPickerToShow: String
    //@State var riskFactorName: String = ""
    //let tempPickerList = ["hi", "bye", "bycycle", "bicycle", "bike"]
    let tempPickerList: [String]
    @State var selectedValue = ""

    var body: some View {
        VStack {

            Text( (riskFactor.value ?? "") + " | Dev Value: " + String(riskFactor.numericalRiskValue))

//            TextField(riskFactor.value ?? "", text: $riskFactorName)
//                .textFieldStyle(RoundedBorderTextFieldStyle())
//                .padding()

            Picker("Please choose a color", selection: $selectedValue) {
                        ForEach(tempPickerList, id: \.self) {
                            Text($0)
                        }
               }.padding()

            Text("You selected: \(selectedValue)")
                .padding()


            Button("Update") {
                if !selectedValue.isEmpty {
                    riskFactor.value = selectedValue
                    riskFactor.numericalRiskValue = riskFactorsModel.returnNumericalRiskValue(pickerName: whichPickerToShow, pickerChoice: selectedValue)
                    coreDM.updateRiskFactor()
                    needsRefresh.toggle()
                }
            }.padding()

            Text("ADD MORE INFO ABOUT RISK FACTOR")
                .padding()

        }.navigationBarTitle(riskFactor.name ?? "")
    }
}
```

g. Ticket 5 – Use apple's core data to store data after the app is closed

I have already implemented this ticket in the previous 3 tickets. The app utilises core data for data persistence, allowing data to be held even after the app is closed, and the phone is shut down.

h. Ticket 6 – Ensure data can be loaded back in after the app is closed

I checked that information can be loaded back into the app after it is closed, and the app is restarted. I ensured that the data from before is the same data loaded in after the app is opened again. Everything worked as intended.



i. Sprint Review

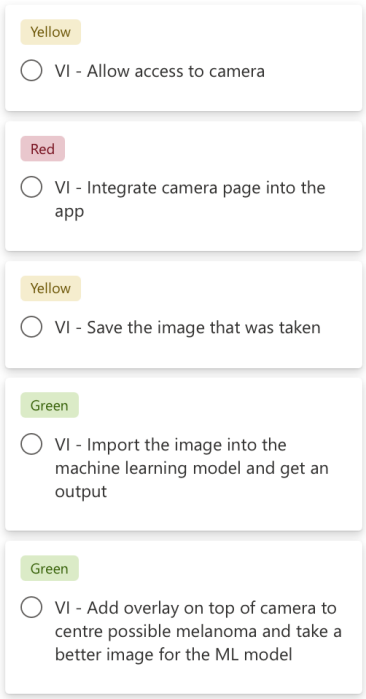
In this sprint I created a system through which I could store data even when the app is closed and the device it is running on is turned off. The testing for individual tickets was done throughout the sprint (such as in ticket 6). Again, I let 5 people use the risk factors page to try break it. Fortunately, my testing throughout the sprint managed to catch any errors that might have occurred. They did mention that I should change the reset button to have a confirmation before resetting all risk factors (which I will do in the aesthetics sprint),

This sprint has worked on the following points specified in the design section:

- [1.a.4] Risk Factors
- [1.a.5] Self diagnosis

7. Sprint 6 - Camera

a. Ticket overview



b. Ticket 1 – Allow access to camera

Apple requires the app to request access to the camera before the app can access that piece of hardware. Therefore, I had to ask for access to the camera in the app's "Information Property List". I added a message along with my request saying, "The melanoma scanner requires access to the camera to take photos of moles in app". This will let the user know what the app needs access to the camera for.

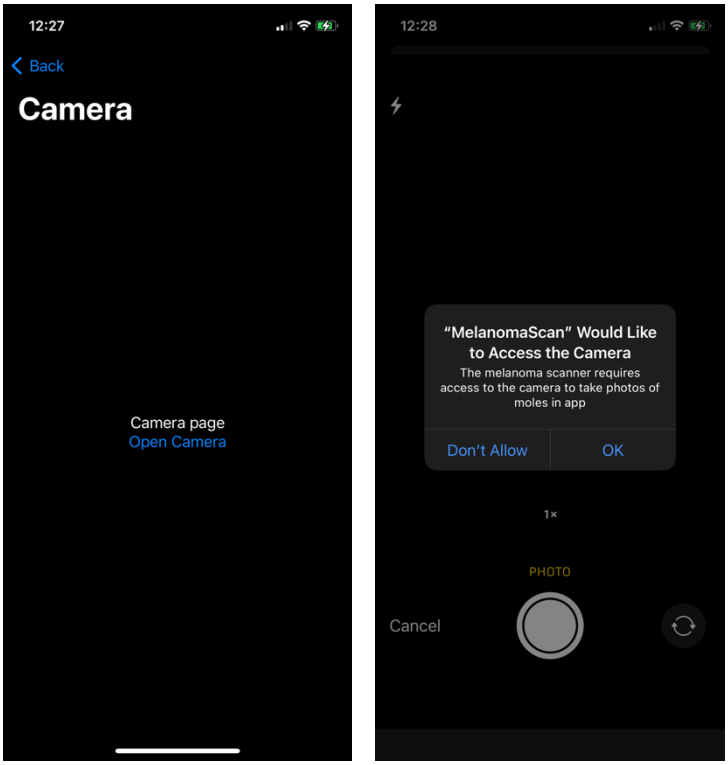
Key	Type	Value
Information Property List	Dictionary	(16 items)
Privacy - Camera Usage Description	String	The melanoma scanner requires access to the camera to take photos of moles in app
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(1 item)
Application supports indirect input events	Boolean	YES
Launch Screen	Dictionary	(0 items)
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)

c. Ticket 2 – Integrate the camera page into the app

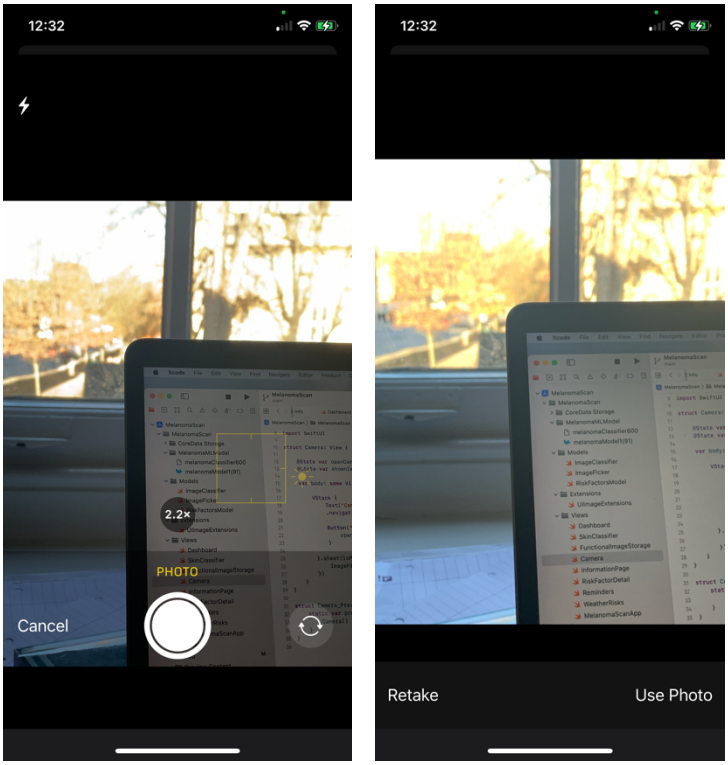
Fortunately, I had already created much of the backend required to use the camera when I implemented the image picker from the camera roll. To make sure that everything would work as I intended it to, I added a button in the camera page that should open the camera, using the functions that I created in sprint 4.

The code for basic camera access is shown below, along with screenshots of what the UI for the code looks like.

```swift
import SwiftUI

struct Camera: View {

    @State var openCamera = false
    @State var shownImage = UIImage()

    var body: some View {

        VStack {
            Text("Camera page")
            .navigationBarTitle("Camera")

            Button("Open Camera") {
                openCamera = true
            }

        }.sheet(isPresented: $openCamera, content: {
            ImagePicker(selectedImage: $shownImage, sourceType: .camera)
        })
    }
}

struct Camera_Previews: PreviewProvider {
    static var previews: some View {
        Camera()
    }
}
```



Once the camera page is opened for the first time, the app will request for camera access as shown above.

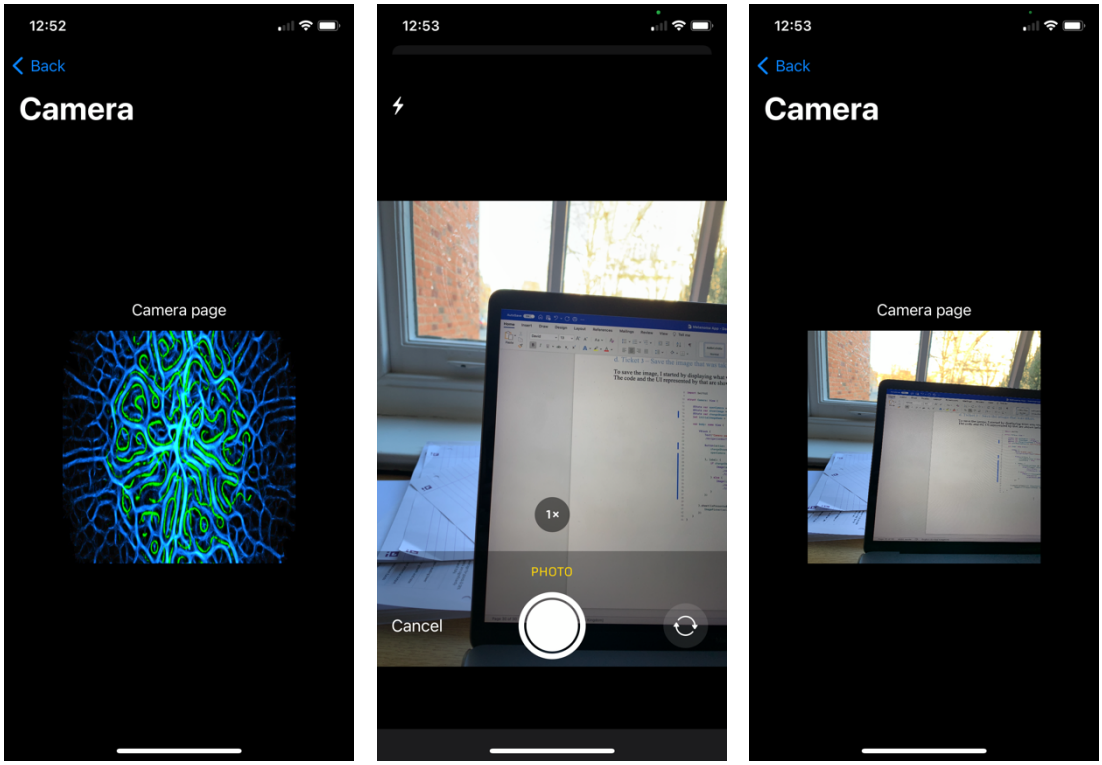


The camera interface allows the user to focus, zoom, adjust lighting, and also retake photos if they did not turn out as expected.

d. Ticket 3 – Save the image that was taken.

To save the image, I started by displaying what was returned from the camera once a photo was taken. The code and the UI represented by that are shown below.

```
import SwiftUI

struct Camera: View {

    @State var openCamera = false
    @State var shownImage = UIImage()
    @State var changeShownImage = false
    let initialImageName = "initialImage"

    var body: some View {

        VStack {
            Text("Camera page")
            .navigationBarTitle("Camera")

            Button(action: {
                changeShownImage = true
                openCamera = true

            }, label: {
                if changeShownImage == true {
                    Image(uiImage: shownImage)
                        .resizable()
                        .frame(width: 250, height: 250)
                } else {
                    Image(initialImageName)
                        .resizable()
                        .frame(width:250,height:250)
                }
            })


        }.sheet(isPresented: $openCamera, content: {
            ImagePicker(selectedImage: $shownImage, sourceType: .camera)
        })
    }
}
```



At this point I realised that having another page for the camera is unnecessary and increases the complexity for both me, and the user for no reason. Therefore, I decided to incorporate the camera aspect of the app in the skin classifier view. I did this by giving the user an option to either use the camera or choose from the photo library when choosing pictures. The code for this is shown below –

```
            Button(action: {

                showChooseCameraOrRollSheet = true

            }, label: {
                if changeClassificationImage == true {
                    Image(uiImage: imageSelectedFromCameraRoll)
                        .resizable()
                        .frame(width: 250, height: 250)
                } else {
                    Image(initialImageName)
                        .resizable()
                        //.frame(width:UIScreen.main.bounds.width*(3/4), height:UIScreen.main.bounds.height*(1/4))
                        .frame(width:250,height:250)
                }
            }).actionSheet(isPresented: $showChooseCameraOrRollSheet) {
                ActionSheet(title: Text("Select Photo"),
                            message: Text("Choose"),
                            buttons: [
                                .default(Text("Photo Library")) {
                                    changeClassificationImage = true
                                    openCameraRoll = true
                                    sourceTypeChoice = .photoLibrary
                                },
                                .default(Text("Camera")) {
                                    changeClassificationImage = true
                                    openCameraRoll = true //make sure cameropen
                                    sourceTypeChoice = .camera
                                },
                                .cancel()
                            ])
            }
```

Then I removed the camera page, and all references to it. Next, I implemented a button that allows the image currently displayed on the skin classifier view to be saved to the user's camera roll. I did this by the following code.

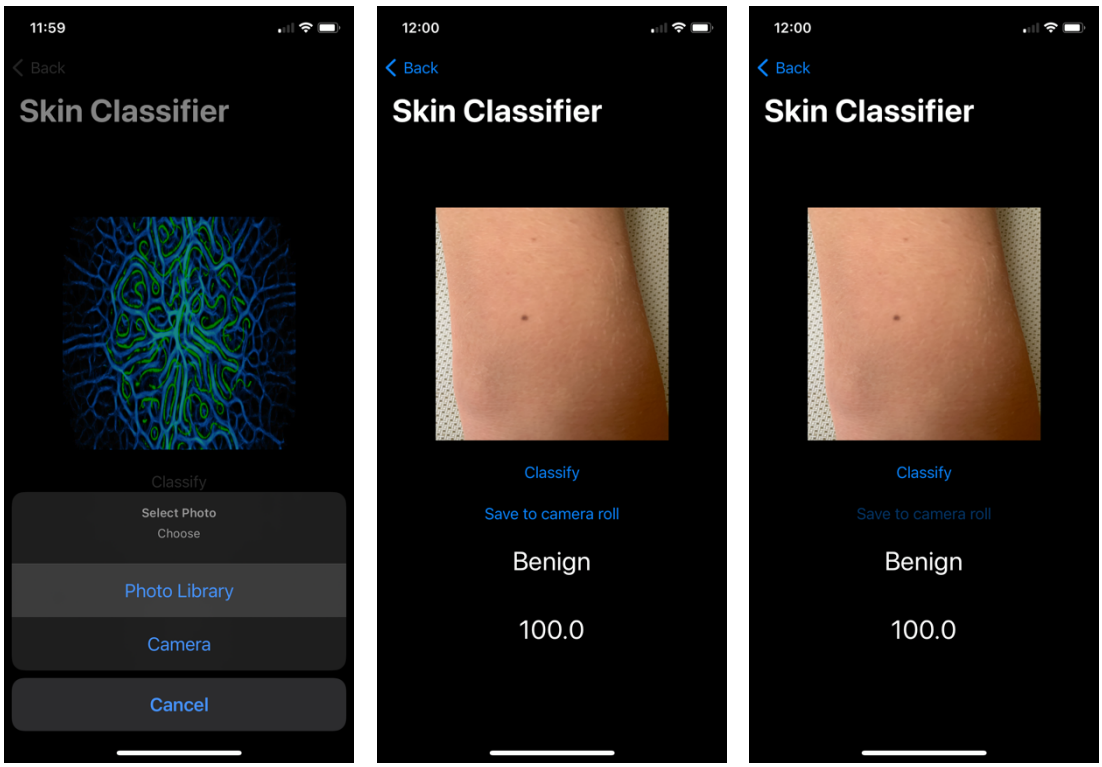
```
87
88              Button("Save to camera roll") {
89
90                  if changeClassificationImage == true {
91                      imageSaver.writeToPhotoAlbum(image: imageSelectedFromCameraRoll)
92                  } else {
93                      imageSaver.writeToPhotoAlbum(image: UIImage(imageLiteralResourceName: initialImageName))
94                  }
95              }
```

```
8  import Foundation
9  import UIKit
10
11 class ImageSaver: NSObject {
12     func writeToPhotoAlbum(image: UIImage) {
13         UIImageWriteToSavedPhotosAlbum(image, self, #selector(saveError), nil)
14     }
15
16     @objc func saveError(_ image: UIImage, didFinishSavingWithError error: Error?, contextInfo: UnsafeRawPointer) {
17         print("Save finished!")
18     }
19 }
20
```

I also had to make sure that the user enables access to the photo library, and I did that by including the correct request in the information property list (with the appropriate message).

Key	Type	Value
Information Property List	Dictionary	(17 items)
Privacy - Photo Library Additions Usage Description	String	The melanoma scanner requires access to the photo library in order to save images you take
Privacy - Camera Usage Description	String	The melanoma scanner requires access to the camera to take photos of moles in app
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(1 item)
Application supports indirect input events	Boolean	YES
Launch Screen	Dictionary	(0 items)
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)

The app's UI now is shown below –



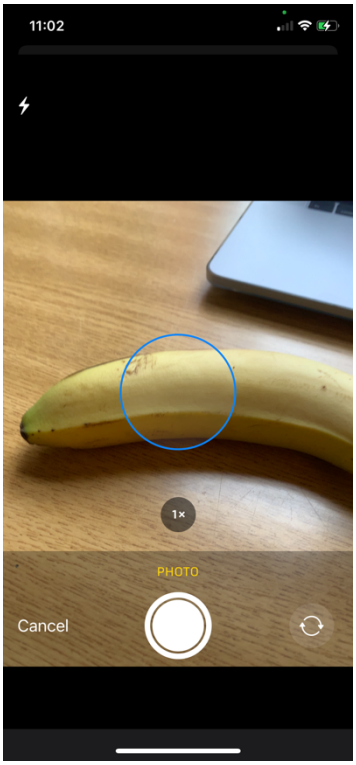
d. Ticket 4 – Import the image into the machine learning model and get an output

I had already implemented this ticket whilst working on the previous one, and so I just tested everything worked by taking multiple photos and making sure that the classifier still worked. Testing of this showed that everything functioned as intended.

e. Ticket 5 – Add overlay on top of camera to centre possible melanoma and take a better image for the ML model

To do this I started by drawing a circle on top of the camera view and making the circle transparent. This allowed me to make the edges of the circle opaque, and so create an area on the screen in which the mole should lie, to aid the user. The code and what the view looked like are shown below.

```
117                 Rectangle()
118                     .fill(Color.white.opacity(0.1))
119                     .overlay(Circle().stroke(Color.blue, lineWidth: 4))
120                     .frame(width: 125, height: 125)
121                     .clipShape(Circle())
```



Whilst testing this, I found a problem – which was that I was unable to click on the screen to focus through the circle. This happened because the circle was treated as an object on top of the camera view, so the tap was only registering on the circle, not on the camera view. Unfortunately, due to the way I created the camera view (using a popup sheet), I could not use apple's recommended method of setting "allowHitTesting to false", which would pass the tap through the object to the background. This didn't work as the tap would just go to the skin classifier view and press buttons such as classify or save image instead.

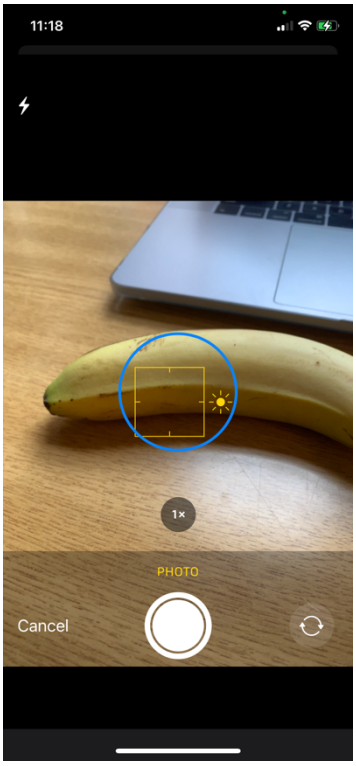
Instead, what I had to do was create a function to draw just the edge of a circle (this is not inbuilt into any apple library. The code for the circle edge creation is shown below.

```swift
import Foundation
import SwiftUI

//draws the edges of a circle for certain amount of angle (in degrees)
struct Arc: Shape {
    var startAngle: Angle
    var endAngle: Angle
    var clockwise: Bool

    func path(in rect: CGRect) -> Path {
        var path = Path()
        path.addArc(center: CGPoint(x: rect.midX, y: rect.midY), radius: rect.width / 2, startAngle: startAngle, endAngle: endAngle, clockwise:
            clockwise)

        return path
    }
}
```

Then I added it to the sheet view including the camera using the following code. Here I had to ensure that the circle didn't pop up when the user wanted to open the photo library (as both the views use the same infrastructure).

```swift
            }.sheet(isPresented: $openCameraRoll, content: {
                ZStack {



                if sourceTypeChoice != .photoLibrary {

                    ImagePicker(selectedImage: $imageSelectedFromCameraRoll, sourceType: sourceTypeChoice)

                    Arc(startAngle: .degrees(0), endAngle: .degrees(360), clockwise: true)
                        .stroke(.blue, lineWidth: 3)
                        .frame(width: 125, height: 125)

                } else {
                    ImagePicker(selectedImage: $imageSelectedFromCameraRoll, sourceType: sourceTypeChoice)
                }

                }
            })
```

The final look of the camera view is shown below, now with the user being able to focus the camera by clicking through the circle.



f. Sprint Review

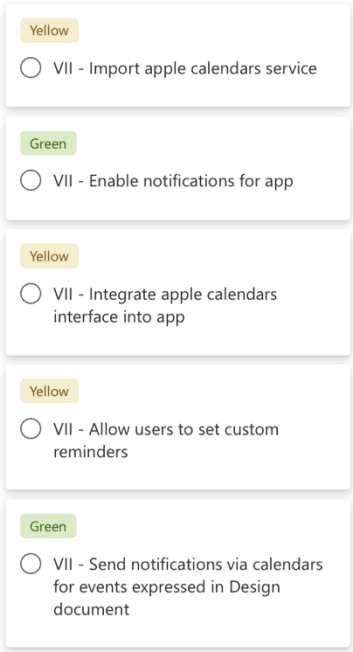
In this sprint I implemented the camera feature for the app. Ticket 5 was added to this sprint as one of the 5 people reviewing my sprint (by attempting to break the app) mentioned that it was not clear where the mole should be when taking a picture. Furthermore, the same error as before occurred where if the user exited the camera without taking a picture there would be no image where there should be one. Fortunately, did this not cause any app crashing errors even when the user tried to classify the image (the placeholder image was being used in this case). I also decided to reach out to Dr Lyman at this point, who mentioned that he liked the direction the app was going in. He mentioned that no matter how good the functionality of the app is, it needs to be user friendly and aesthetically pleasing if I want users to actually use it properly rather than just to test it out.

This sprint has worked on the following points specified in the design section:

- [1.a.1] Skin Classifier
- [1.a.7] Camera

8. Sprint 7 - Calendar

a. Ticket overview



b. Ticket 1 – Import Apple calendars service

To implement reminders, I decided to use the pre-built calendars and reminders service apple provides, called “EventKit”. I imported it into the app, and created a new object called RemindersManager, which would store all the methods required to deal with reminders.

c. Ticket 2 – Enable notifications for app

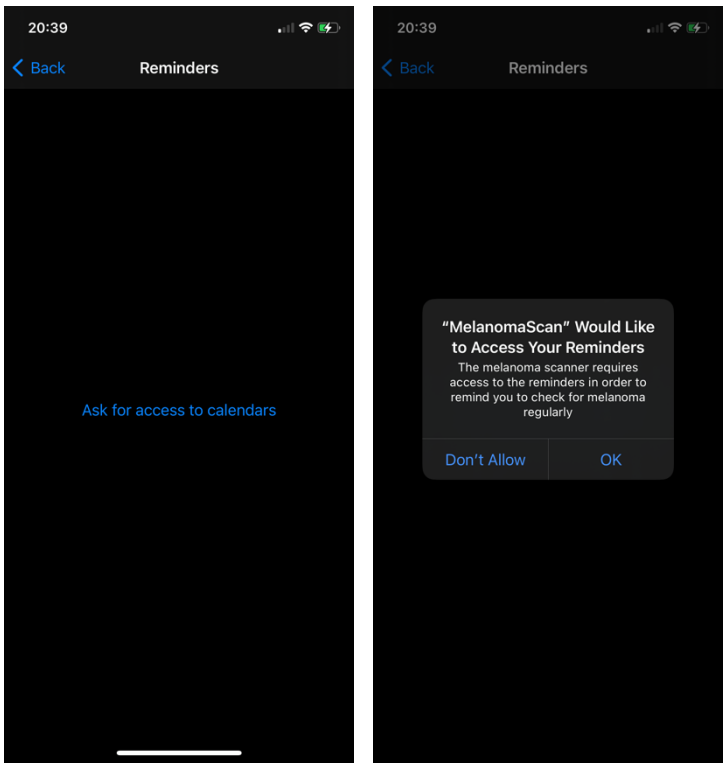
There was very little documentation about this found online, and so I did not realise at first that reminders and calendars required separate permission to be accessed. This caused me to run into errors in which instead of creating a reminder the app would print the error shown below.

To fix this I created a method inside my RemindersManager class to request access to the reminders app on an iphone. As well as this I had to add information to the info.plist file describing what the user should be shown when asked to provide access to a certain component of the phone. The code for the request access method is shown below.

```
28      func requestRemindersAccess() {
29
30          store.requestAccess(to: .reminder, completion:
31            {(granted: Bool, error: Error?) -> Void in
32                if granted {
33                  //self.insertEvent(store: store)
34                  print("Access granted")
35                } else {
36                  print("Access denied")
37                }
38          })
39      }
```

Then I created a button on the reminders page of the app that requests access from the user. The code, and the UI for that code are shown below. Now the app, with the user's permission, can make use of apple's reminders service.

```
18              Button("Ask for access to calendars") {
19                  remindersManager.requestRemindersAccess()
20              }
```



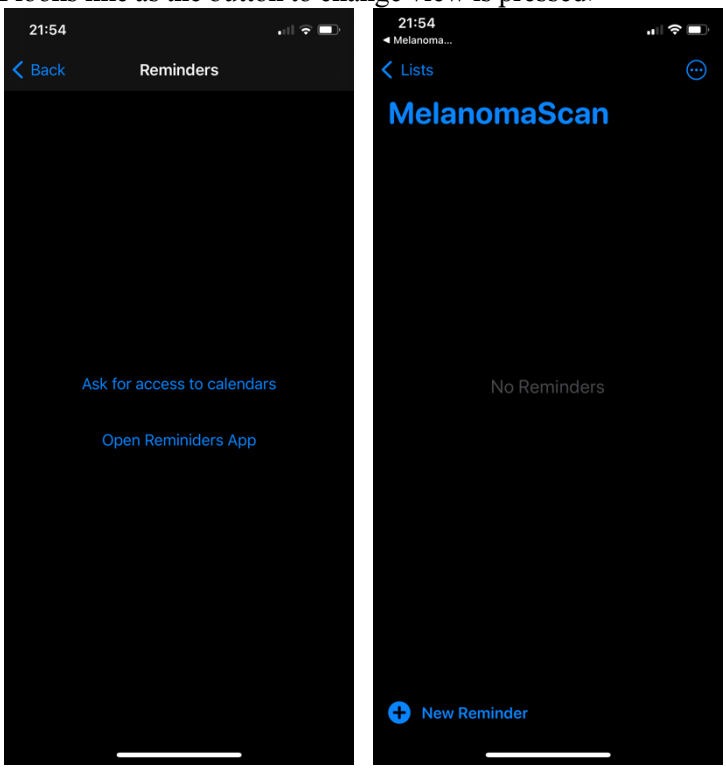
d. Ticket 3 – Integrate apple calendars interface into app

After some research I found out that it is impossible (at the time of writing this) to add another 3rd party app's (even if it is apple) views into your apps views. Therefore, it is impossible for me to add the calendars interface into my app. The best I could do is create a button that links to apple's reminders app, opening the reminders that the melanoma app has created. The code to create a link and open the reminders app is shown below.

```
72      func openRemindersApp() {
73          if let url = URL(string: "x-apple-reminderkit://MelanomaScan"), UIApplication.shared.canOpenURL(url) {
74              UIApplication.shared.open(url, options: [:]) { (isDone) in
75
76              }
77          }
78      }
```

```
26              Button("Open Reminiders App") {
27                  remindersManager.openRemindersApp()
28              }.padding()
```

Below is what the UI looks like as the button to change view is pressed.



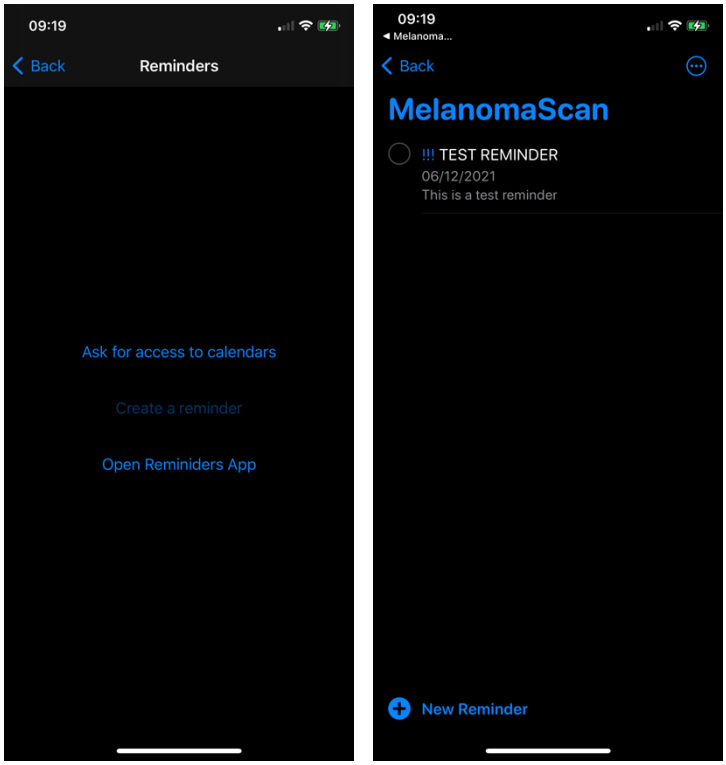
d. Ticket 4 – Allow users to set custom reminders

First, I started off by creating a function that would create and save a reminder for a certain date and time. Then I created a button to call that function from the reminders app. The code for this is shown below, along with what happens when the button is pressed.

```
42      // function to create a new reminder (pass in title, description and how long after today the reminder should be set for
43      func createAndSaveNewReminder(calendarTitle: String, calendarNotes: String, timeInterval: Double) {
44          guard let calendar = self.store.defaultCalendarForNewReminders() else {
45              print("Default calendar not created")
46              return
47          }
48
49          print("working here")
50          print(calendar)
51
52          let newReminder = EKReminder(eventStore: store)
53          newReminder.calendar = calendar
54          newReminder.title = calendarTitle
55
56          newReminder.priority = Int(EKReminderPriority.high.rawValue)
57          newReminder.notes = calendarNotes
58
59          let dueDate = Date().addingTimeInterval(timeInterval)
60          newReminder.dueDateComponents = Calendar.current.dateComponents([.year, .month, .day], from: dueDate)
61
62          do {
63              print("place1")
64              try self.store.save(newReminder, commit: true)
65              print("place2")
66          } catch let error {
67              print("place3")
68              print(error)
69          }
70
71      }
```

```
22              Button("Create a reminder") {
23                  remindersManager.createAndSaveNewReminder(calendarTitle: "TEST REMINDER", calendarNotes: "This is a test reminder", timeInterval: Double(60 * 60 * 24 * 3) )
24              }.padding()
```

The button shown above should now set a reminder called “TEST REMINDER” for 3 days after today. Testing if this works as expected –



At this point I decided to incorporate the request to access calendars into the "create a reminder" button, so that the user does not have to go out of their way to provide access, instead access will be requested anytime the user tries to create a reminder. This will improve usability and reduce the likelihood of an error due to access to calendars not being granted.

Then I decided to allow the user to decide the frequency of reminders they wanted. I did this by letting them choose from a variety of values using a picker.

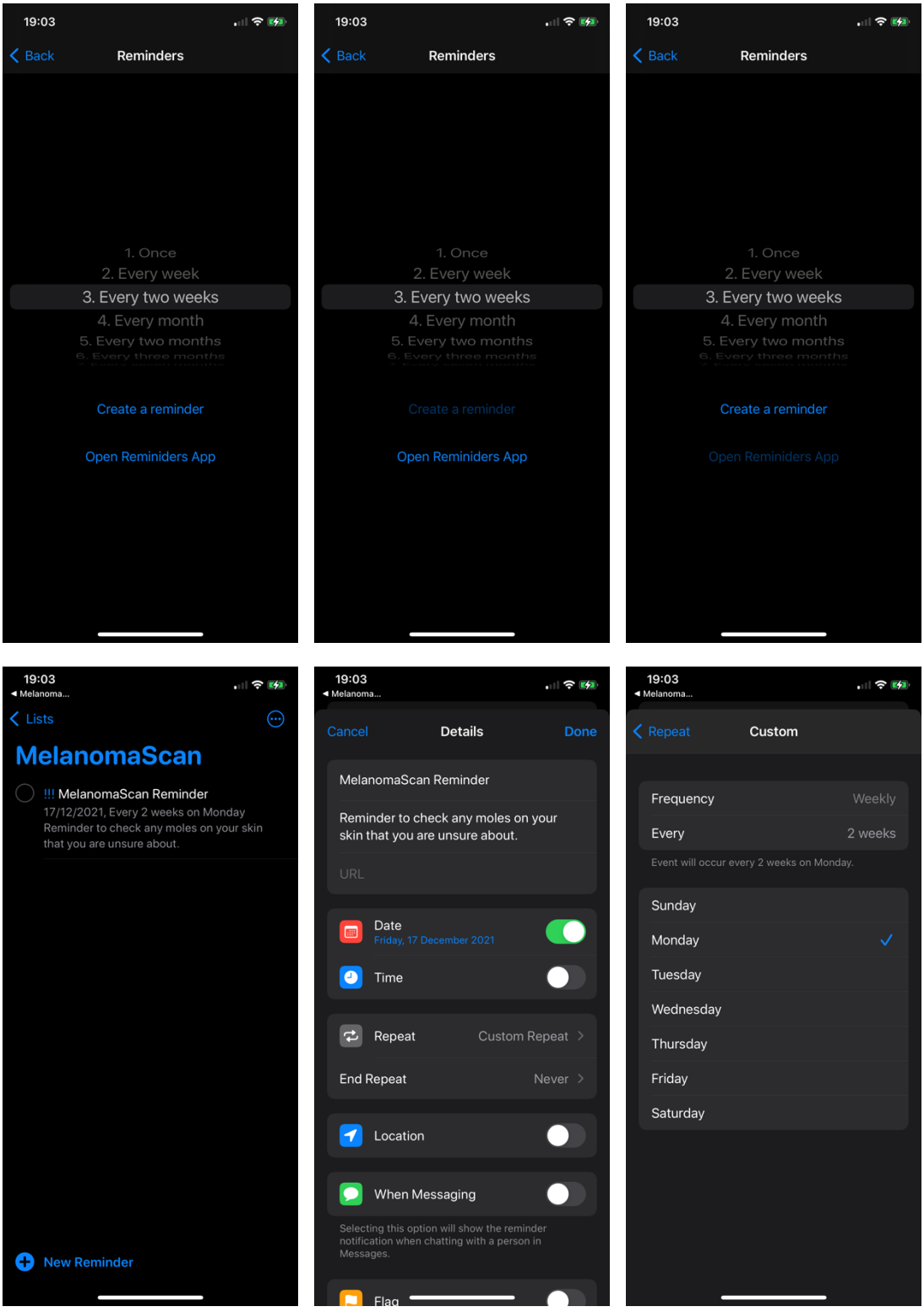
```
15      @State var selectedTimeIntervalKey = "Every week"
16      @State var selectedTimeIntervalValue = 7
17      let reminderTimeIntervalDictionary =   ["1. Once": -1,
18                                              "2. Every week": 7,
19                                              "3. Every two weeks": 14,
20                                              "4. Every month": 30,
21                                              "5. Every two months": 60,
22                                              "6. Every three months": 90,
23                                              "7. Every seven months": 180]
24
25      var body: some View {
26
27          VStack {
28
29              //pickers so that the user can decide how often and for how long reminders appear.
30
31              Picker(selection: $selectedTimeIntervalKey, label: Text("Reminder Interval")) {
32                  ForEach(reminderTimeIntervalDictionary.sorted(by: <), id: \.key) { key, val in
33                      Text(key)
34                  }
35              }
36
37              Button("Create a reminder") {
38                  //method goes here to use what user inputted.
39                  remindersManager.userCreateAndSaveNewReminder(calendarTitle: "MelanomaScan Reminder", calendarNotes: "Reminder to check any moles on your skin that you are unsure about.",
                        timeInterval: reminderTimeIntervalDictionary[selectedTimeIntervalKey] ?? -1)
40              }.padding()
41
```

As you can see above, I also changed the subroutine called when the user clicks the "Create a reminder" button. I created a new subroutine that takes in the user's input and manipulates it into the format required by the EKEvent library. Then (shown in the code below), I called the method I created earlier to create the reminder. I also had to change parts of the original create reminder function, as previously I had not incorporated a way to cause a reminder to repeat.

```swift
    // creates a reminder using the user's inputs.
    func userCreateAndSaveNewReminder(calendarTitle: String, calendarNotes: String, timeInterval: Int) {
        //deal with special case of "once"
        if timeInterval == -1 {
            self.createAndSaveNewReminder(calendarTitle: calendarTitle, calendarNotes: calendarNotes, timeInterval: Double(60 * 60 * 24 * timeInterval), numWeeksPerRepeat: -1)
        } else {

            //convert from repeat (for how many days), to value for how many weeks.
            var numWeeksPerReminder: Int = 0
            numWeeksPerReminder = timeInterval/7
            print("**************")
            print(numWeeksPerReminder)
            print("**************")

            self.createAndSaveNewReminder(calendarTitle: calendarTitle, calendarNotes: calendarNotes, timeInterval: Double(60 * 60 * 24 * timeInterval), numWeeksPerRepeat: numWeeksPerReminder)

        }


    }
```

```swift
   // function to create a new reminder (pass in title, description and how long after today the reminder should be set for
   func createAndSaveNewReminder(calendarTitle: String, calendarNotes: String, timeInterval: Double, numWeeksPerRepeat: Int) {

       if self.accessGranted == false {

           self.requestRemindersAccess()

       } else {

           guard let calendar = self.store.defaultCalendarForNewReminders() else {
               print("Default calendar not created")
               return
           }

           print("working here")
           print(calendar)

           let newReminder = EKReminder(eventStore: store)
           newReminder.calendar = calendar
           newReminder.title = calendarTitle

           newReminder.priority = Int(EKReminderPriority.high.rawValue)
           newReminder.notes = calendarNotes

           let dueDate = Date().addingTimeInterval(timeInterval)
           newReminder.dueDateComponents = Calendar.current.dateComponents([.year, .month, .day], from: dueDate)

           if numWeeksPerRepeat == -1 {

               do {
                   try self.store.save(newReminder, commit: true)
                   print("** Reminder Saved **")
               } catch let error {
                   print(error)
               }

           } else {

               let recurrenceRule = EKRecurrenceRule(recurrenceWith: .weekly, interval: numWeeksPerRepeat, daysOfTheWeek: [EKRecurrenceDayOfWeek(.monday)], daysOfTheMonth: nil, monthsOfTheYear: nil, weeksOfTheYear: nil, daysOfTheYear:
                   nil, setPositions: nil, end: nil)
               newReminder.addRecurrenceRule(recurrenceRule)

               do {
                   try self.store.save(newReminder, commit: true)
                   print("** Reminder Saved **")
               } catch let error {
                   print(error)
               }

           }

       }
   }
```

The code's functionality is shown in the following screenshots. The screenshots show how in the reminders app a 'repeat' value is added, causing the reminder to be repeated by the value specified by the user.



e. Ticket 5 – Send notifications via calendars for events expressed in Design document

The reminders created via the EKEvent library are directly linked to apple's calendars and reminders apps. Therefore, notifications can be set up (and have been set up in the previous tickets), but will be sent through apple's own apps. Due to the apps being apple's own apps I can be sure that the notifications will not be blocked/turned off.

f. Sprint Review

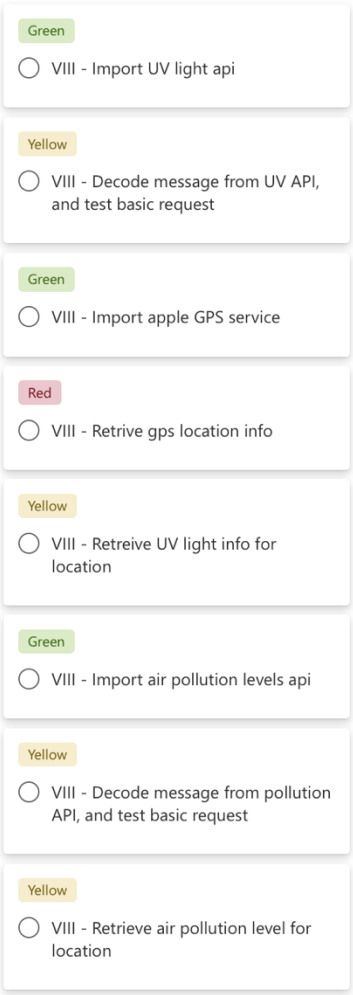
In this sprint I allowed the app to access apple's reminders feature and create repeating reminders for the user to check their skin. At first, I attempted to implement my own reminders service but I found that was too unreliable given the user can close the app and shut down their phone which pauses the timer feature I was using. The 5 people I asked to test this feature also managed to cause the app to crash by creating two reminders for the same time. This led me to using apple's reminders feature to create more reliable reminders (which did not cause any errors in post sprint testing).

This sprint has worked on the following points specified in the design section:

- [1.a.6] Reminders

9. Sprint 8 – Environment Risks

a. Ticket overview



b. Ticket 1 – Import UV light API

I decided to use the openUV api (https://www.openuv.io), as it was an api that allowed me to access UV index levels for anywhere in the world, at any given time. The api required me to create an authorisation key (by signing up to their service. Unfortunately, their free service is limited to 50 requests per day, but that was more than sufficient for the development of my app.

I set up everything for the UV light api requests by creating a new file called UVIndexManager. This file will contain everything from requesting and decoding the JSON, to carrying out algorithmic tasks with the data from the openUV api.

I started off by creating the data structures that would hold data from the returned JSON. The JSON request format is shown below, and the data structures I created to hold the data I wanted from the request are shown below as well.


```
{
  - result: {
        uv: 0,
        uv_time: "2021-12-04T11:58:40.741Z",
        uv_max: 12.6724,
        uv_max_time: "2021-12-04T04:08:54.537Z",
        ozone: 291.2,
        ozone_time: "2021-12-04T09:04:14.502Z",
      - safe_exposure_time: {
            st1: null,
            st2: null,
            st3: null,
            st4: null,
            st5: null,
            st6: null
        },
      - sun_info: {
          - sun_times: {
                solarNoon: "2021-12-04T04:08:54.537Z",
                nadir: "2021-12-03T16:08:54.537Z",
                sunrise: "2021-12-03T21:06:41.820Z",
                sunset: "2021-12-04T11:11:07.254Z",
                sunriseEnd: "2021-12-03T21:09:29.652Z",
                sunsetStart: "2021-12-04T11:08:19.422Z",
                dawn: "2021-12-03T20:39:03.656Z",
                dusk: "2021-12-04T11:38:45.418Z",
                nauticalDawn: "2021-12-03T20:05:27.702Z",
                nauticalDusk: "2021-12-04T12:12:21.372Z",
                nightEnd: "2021-12-03T19:29:26.888Z",
                night: "2021-12-04T12:48:22.186Z",
                goldenHourEnd: "2021-12-03T21:41:56.807Z",
                goldenHour: "2021-12-04T10:35:52.267Z"
            },
          - sun_position: {
                azimuth: 0.9819919359817836,
                altitude: -0.16995471961916217
            }
        }
    }
}
```


```
10
11  struct Root_Layer0: Codable {
12      var result: Result_Layer1
13  }
14
15  struct Result_Layer1: Codable {
16      var uv: Double
17      var uv_max: Double
18      var ozone: Double
19  }
20  // May also add sunrise and sunset times soon
```

The structs shown above only specify the data I wanted from the request, and the rest is discarded.

c. Ticket 2 – Decode message from UV API, and test basic request

In the file that includes the data structures of the decoded json I added a class called UVIndexManager, which will include all subroutines that carry out algorithms on the data requested from the openUV api.

I first created a method called test request which sends out a set request to the openUV api and stores the results in the structs I defined earlier. Then it returns the values in the structs so that I can use them and display them for the user. Below is the method that sends a fixed request to test everything is working as intended.

```
30      func testRequest(callback: @escaping (Double, Double, Double) -> ()) {
31          //defining what the request is
32          let request = NSMutableURLRequest(url: NSURL(string:
                "https://api.openuv.io/api/v1/uv?lat=-33.34&lng=115.342&dt=2018-01-24T10%3A50%3A52.283Z")! as URL, cachePolicy:  .useProtocolCachePolicy,
                timeoutInterval: 10.0)
33
34          request.httpMethod = "GET"
35          request.allHTTPHeaderFields = headers
36
37          let session = URLSession.shared
38          let dataTask = session.dataTask(with: request as URLRequest, completionHandler: { (data: Data?, response: URLResponse?, error:  Error?) ->
                Void in
39              //checking if response actually contains data
40              guard let data = data, error == nil else {
41                  if let printError = error {
42                      print(printError)
43                  } else {
44                      print("error in printError")
45                  }
46                  return
47              }
48
49              var result: Root_Layer0?
50              //decoding the JSON returned
51              do {
52                  let decoder = JSONDecoder()
53                  decoder.dateDecodingStrategy = .iso8601
54                  result = try decoder.decode(Root_Layer0.self, from: data)
55              }
56              catch {
57                  print("-----ERROR-----")
58                  print(error)
59                  print("-----ERROR-----")
60              }
61              guard let json = result else{
62                  return
63              }
64              //returning the decoded values stored in structs.
65              DispatchQueue.main.async {
66                  callback(json.result.uv, json.result.uv_max, json.result.ozone)
67              }
68
69          })
70
71          dataTask.resume()
72      }
```

I called this method from the UI, and printed the values returned.

```
18              Button("Test Request") {
19                  uvIndexManager.testRequest(callback: {(uvResponse: Double, uvMaxResponse: Double, ozoneResponse: Double) -> () in
20                      print("UV Index: " + String(uvResponse))
21                      print("Max UV Index: " + String(uvMaxResponse))
22                      print("Ozone levels: " + String(ozoneResponse))
23                  } )
24              }
```

This code prints the following data, showing that the fixed request is working correctly.

```
UV Index: 0.1234
Max UV Index: 11.9421
Ozone levels: 293.4
```

I also made a separate request earlier, using an earlier iteration of the fixed request method (shown below). This method printed the http response (also shown below), which made it clear that the api requests were working exactly as intended.

```swift
class UVIndexManager {

    //using the openUV api

    let headers = [
      "x-access-token": "5dc7e2dde16e30b5ad087bc0fe9577cd"
    ]

    //function to do a test request from the openUV api
    func testRequest() {

        let request = NSMutableURLRequest(url: NSURL(string: "https://api.openuv.io/api/v1/uv?lat=-33.34&lng=115.342&dt=2018-01-24T10%3A50%3A52.283Z")!
            as URL, cachePolicy: .useProtocolCachePolicy, timeoutInterval: 10.0)

        request.httpMethod = "GET"
        request.allHTTPHeaderFields = headers

        let session = URLSession.shared
        let dataTask = session.dataTask(with: request as URLRequest, completionHandler: { (data, response, error) -> Void in
         if (error != nil) {
             if let printError = error {
                 print(printError)
             } else {
                 print("error in printError")
             }
           //print(error)
         } else {
           let httpResponse = response as? HTTPURLResponse
             if let printResponse = httpResponse {
                 print(printResponse)
             } else {
                 print("error in httpResponse")
             }
           //print(httpResponse)
         }
        })

        dataTask.resume()
    }

}

```

```
<NSHTTPURLResponse: 0x283aa6960> { URL:
    https://api.openuv.io/api/v1/uv?lat=-33.34&lng=115
    .342&dt=2018-01-24T10%3A50%3A52.283Z } { Status Code: 200, Headers {
    "Access-Control-Allow-Origin" =     (
        "*"
    );
    Connection =     (
        "keep-alive"
    );
    "Content-Length" =     (
        902
    );
    "Content-Type" =     (
        "application/json; charset=utf-8"
    );
    Date =     (
        "Sat, 04 Dec 2021 12:13:24 GMT"
    );
    Etag =     (
        "W/\"386-z9aDIqnH/BAUXnh+oX76gsk10i8\""
    );
    Server =     (
        Cowboy
    );
    Via =     (
        "1.1 vegur"
    );
    "X-Powered-By" =     (
        Express
    );
    "X-Ratelimit-Limit" =     (
        50
    );
    "X-Ratelimit-Remaining" =     (
        48
    );
} }
```

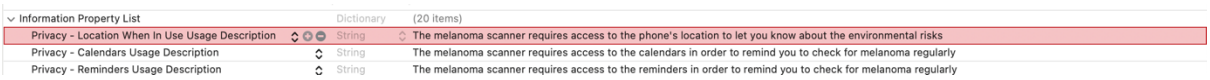
d. Ticket 3 – Import Apple GPS service

To send the correct longitude and latitude for the api request, I had to get the user's co-ordinates. Apple's iPhones have inbuilt gps software and hardware, so I decided to make use of that. After some research I found that the apple's CoreLocation library was the best for me to use.

I created a new file called "LocationManager" to deal with all the location requests and related algorithms. I set it up using the code shown below, initialising everything required by the CoreLocation library. I made the class inside the file an observable object, so that any updates in values there would also cause text in the UI to be updated. Furthermore, I had to make it into a CLLocationManagerDelegate as required by Core Location.

```swift
import Foundation
import CoreLocation
import Combine

class LocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {

    //setting up everything required from CoreLocation library
    //set up as published, so functions do not need to return values
    private let locationManager = CLLocationManager()
    @Published var locationStatus: CLAuthorizationStatus?
    @Published var lastLocation: CLLocation?

    //required setting up of CoreLocation instance
    override init() {
        super.init()
        locationManager.delegate = self
        locationManager.desiredAccuracy = kCLLocationAccuracyBest
        locationManager.requestWhenInUseAuthorization()
        locationManager.startUpdatingLocation()
    }
```

I also had to add a description of why the location was being used in the info.plist file.



e. Ticket 4 – Retrieve gps location info

To retrieve the user's location, I had to first ensure that I was authorized, and so also wanted to let the user know if they had enabled location access. Then I created functions to constantly update the user's displayed location. The code for this is shown below.

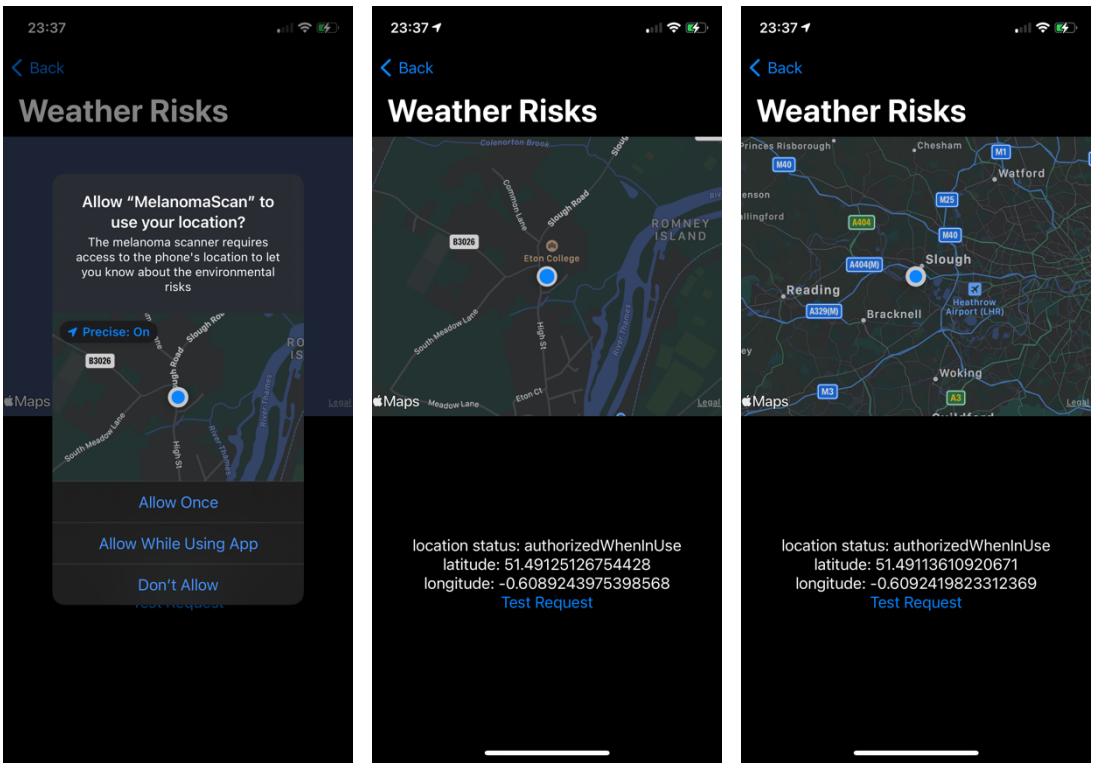
```swift
    //checks whether user has allowed access to the location services
    //if the user has not given acess, returns what access given
    var statusString: String {
        guard let status = locationStatus else {
            return "unknown"
        }
        switch status {
        case .notDetermined: return "notDetermined"
        case .authorizedWhenInUse: return "authorizedWhenInUse"
        case .authorizedAlways: return "authorizedAlways"
        case .restricted: return "restricted"
        case .denied: return "denied"
        default: return "unknown"
        }
    }

    //updating the location status
    func locationManager(_ manager: CLLocationManager, didChangeAuthorization status: CLAuthorizationStatus) {
        locationStatus = status
        //print(#function, statusString)
    }

    //updating the current location
    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
        guard let location = locations.last else { return }
        lastLocation = location
        //print(#function, location)
    }
```

I also decided to add a small map showing the user's current location to the view. This will make the location more tangible and obvious to the user, making it clearer what the information being presented in the view is for. It will also allow the user to know exactly for what area the UV/pollution risk is high/low for.

I implemented the map using apple's Map Kit, and made the map constantly centred around the user. I also added text in the view to show the user's longitude and latitude, as well as access to location data status (for development purposes). The code for these UI features are shown below.

```swift
import SwiftUI
import MapKit

struct WeatherRisks: View {

    let uvIndexManager = UVIndexManager()

    @StateObject var locationManager = LocationManager()

    //setting the user's co-ordinates
    var userLatitude: String {
        return "\(locationManager.lastLocation?.coordinate.latitude ?? 0)"
    }
    var userLongitude: String {
        return "\(locationManager.lastLocation?.coordinate.longitude ?? 0)"
    }

    //using apple's Mapkit to create an initial region, completely random
    @State private var region = MKCoordinateRegion(center: CLLocationCoordinate2D(latitude: 1, longitude: 1), span:
        MKCoordinateSpan(latitudeDelta: 0.5, longitudeDelta: 0.5))

    var body: some View {

        VStack {

            Map(coordinateRegion: $region, showsUserLocation: true, userTrackingMode: .constant(.follow))
                        .frame(width: 400, height: 300)

            Spacer()

            VStack {

                Text("location status: \(locationManager.statusString)")
                Text("latitude: \(userLatitude)")
                Text("longitude: \(userLongitude)")

                Button("Test Request") {
                    uvIndexManager.testRequest(callback: {(uvResponse: Double, uvMaxResponse: Double, ozoneResponse: Double) -> () in
                        print("UV Index: " + String(uvResponse))
                        print("Max UV Index: " + String(uvMaxResponse))
                        print("Ozone levels: " + String(ozoneResponse))
                    } )
                }

            }
        }
```

The corresponding UI is shown below.



f. Ticket 5 – Retrieve UV light info for location

In order to retrieve UV light information for the user's location I started off by converting the api request function into a function that works with passed in values for the user's co-ordinates. I did this by replacing (with string interpolation) the longitude and latitude values in the URL with values given by the user's GPS co-ordinates. The code for the final api request function is shown below.

```
72      //requests UV info for given location and current time. Works Async.
73      func requestUVInfoForLocation(inputLatitude: Double, inputLongitude: Double, callback: @escaping (Double, Double, Double) -> ()) {
74
75          //URL for request
76          let request = NSMutableURLRequest(url: NSURL(string: "https://api.openuv.io/api/v1/uv?lat=" + String(inputLatitude) + "&lng=" +
                String(inputLongitude))! as URL, cachePolicy: .useProtocolCachePolicy, timeoutInterval: 10.0)
77
78          request.httpMethod = "GET"
79          request.allHTTPHeaderFields = headers
80
81          let session = URLSession.shared
82          let dataTask = session.dataTask(with: request as URLRequest, completionHandler: { (data: Data?, response: URLResponse?, error: Error?) ->
                Void in
83              //checking if response actually contains data
84              guard let data = data, error == nil else {
85                  if let printError = error {
86                      print(printError)
87                  } else {
88                      print("error in printError")
89                  }
90                  return
91              }
92
93              var result: Root_Layer0?
94              //decoding the JSON returned
95              do {
96                  let decoder = JSONDecoder()
97                  decoder.dateDecodingStrategy = .iso8601
98                  result = try decoder.decode(Root_Layer0.self, from: data)
99              }
100             catch {
101                 print("-----ERROR-----")
102                 print(error)
103                 print("-----ERROR-----")
104             }
105             guard let json = result else{
106                 return
107             }
108             //returning the decoded values stored in structs, works async.
109             DispatchQueue.main.async {
110                 callback(json.result.uv, json.result.uv_max, json.result.ozone)
111             }
112
113         })
114
115         dataTask.resume()
116     }
117
118 }
```

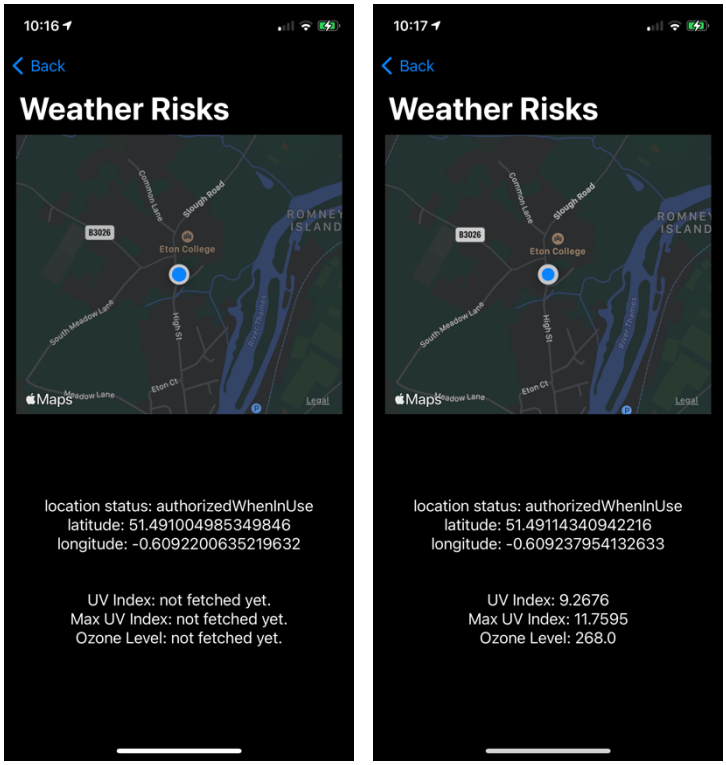
To make the app easier to use, I made the api request happen anytime the user opens the weather risks. This was done using "onAppear", as shown in the code below.

```
57          .onAppear(perform: {
58              uvIndexManager.requestUVInfoForLocation(inputLatitude: Double(userLatitude) ?? 0, inputLongitude: Double(userLongitude) ?? 0,callback: {(uvResponse:
                    Double, uvMaxResponse: Double, ozoneResponse: Double) -> () in
59                  uvIndexDisplay = String(uvResponse)
60                  uvMaxIndexDisplay = String(uvMaxResponse)
61                  ozoneLevelDisplay = String(ozoneResponse)
62              })
63          })
```

As the api request happens asynchronously, I let the user know when the data had not yet been fetched by setting the initial text to "not fetched yet". And then I displayed the information (currently for development purposes), as in the code below.

```
28      @State var uvIndexDisplay = "not fetched yet."
29      @State var uvMaxIndexDisplay = "not fetched yet."
30      @State var ozoneLevelDisplay = "not fetched yet."
31
32      var body: some View {
33
34          VStack {
35
36              Map(coordinateRegion: $region, showsUserLocation: true, userTrackingMode: .constant(.follow))
37                  .frame(width: 350, height: 300)
38
39              Spacer()
40
41              VStack {
42
43                  VStack {
44                      Text("location status: \(locationManager.statusString)")
45                      Text("latitude: \(userLatitude)")
46                      Text("longitude: \(userLongitude)")
47                  }.padding()
48
49                  VStack {
50                      Text("UV Index: " + uvIndexDisplay)
51                      Text("Max UV Index: " + uvMaxIndexDisplay)
52                      Text("Ozone Level: " + ozoneLevelDisplay)
53                  }.padding()
54
55
```

After this ticket the weather risks page looks like the screenshots below.



g. Ticket 6 – Import air pollution levels api

I used the same method as the UV light api to implement the air pollution api. I first found an api that returned sufficient useful data for free. The IQAir AirVisual api did exactly what I needed from the air pollution api, and so I decided to use it (https://www.iqair.com/commercial/air-quality-monitors/airvisual-platform/api).

I added a file called AQIManager, inside which I created a AQIManager class (which I will use to do anything AQI related. Then I modified the UV light api request code so that it worked with the IQAir api, the code for this is shown below.

```swift
class AQIManager {

    let accessKey = "990d95b6-cdee-43c6-b66f-73ceee50d2da"

    //function to do a test request from the openUV api
    func testRequest(callback: @escaping (String, Int) -> ()) {
        //defining what the request is
        let request = NSMutableURLRequest(url: NSURL(string: "http://api.airvisual.com/v2/nearest_city?lat=51.5072&lon=0.1276&key=\(accessKey)")! as
            URL, cachePolicy: .useProtocolCachePolicy, timeoutInterval: 10.0)

        request.httpMethod = "GET"
        //request.allHTTPHeaderFields = headers

        let session = URLSession.shared
        let dataTask = session.dataTask(with: request as URLRequest, completionHandler: { (data: Data?, response: URLResponse?, error: Error?) ->
            Void in
            //checking if response actually contains data
            guard let data = data, error == nil else {
                if let printError = error {
                    print(printError)
                } else {
                    print("error in printError")
                }
                return
            }

            var result: AQI_Root_Layer0?
            //decoding the JSON returned
            do {
                let decoder = JSONDecoder()
                decoder.dateDecodingStrategy = .iso8601
                result = try decoder.decode(AQI_Root_Layer0.self, from: data)
            }
            catch {
                print("-----ERROR-----")
                print(error)
                print("-----ERROR-----")
            }
            guard let json = result else{
                return
            }
            //returning the decoded values stored in structs.
            DispatchQueue.main.async {
                callback(json.data.city, json.data.current.pollution.aqius)
            }
        })
        dataTask.resume()
    }

}
```

h. Ticket 7 – Decode message from pollution API, and test basic request

I also had to take a look at an example response from the api so that I could structure the way I store the json requests I receive. The parts of the response that would be helpful for creating the data structure or would be helpful to use are shown below (the whole response is very long as it contains historic data, as well as data for forecasts).

```
{
  "status": "success",
  "data": {
    "name": "Eilat Harbor",
    "city": "Eilat",
    "state": "South District",
    "country": "Israel",
    "location": {
      "type": "Point",
      "coordinates": [
        34.939443,
        29.531814
      ]
    },
    "forecasts": [ //object containing forecast information
      {
        "ts": "2017-02-01T03:00:00.000Z",  //timestamp
        "aqius": 21, //AQI value based on US EPA standard
        "aqicn": 7, //AQI value based on China MEP standard
        "tp": 8, //temperature in Celsius
        "tp_min": 6, //minimum temperature in Celsius
        "pr": 976,  //atmospheric pressure in hPa
        "hu": 100, //humidity %
        "ws": 3, //wind speed (m/s)
        "wd": 313, //wind direction, as an angle of 360° (N=0, E=90, S=180, W=270)
        "ic": "10n" //weather icon code, see below for icon index
      },
    …  // contains more forecast data for upcoming 76 hours
    ]
    "current": {
      "weather": {
        "ts": "2017-02-01T01:00:00.000Z",
        "tp": 12,
        "pr": 1020,
        "hu": 62,
        "ws": 2,
        "wd": 320,
        "ic": "01n"
      },
      "pollution": {
        "ts": "2017-02-01T01:15:00.000Z",
        "aqius": 18,
        "mainus": "p1", //main pollutant for US AQI
        "aqicn": 20,
        "maincn": "p1",  //main pollutant for Chinese AQI
        "p1": {   //pollutant details, concentration and appropriate AQIs
          "conc": 20,
          "aqius": 18,
          "aqicn": 20
        }
      }
    },
    "history": { //object containing weather and pollution history information
      "weather": [
        {
          "ts": "2017-02-01T01:00:00.000Z",
          "tp": 12,
          "pr": 1020,
          "hu": 62,
          "ws": 2,
          "wd": 320,
          "ic": "01n"
```


Using this information about an example response I created the data structure I will use to store the response. This structure will allow me to return what city the AQI is measured for (as air pollution measuring stations only exist in certain cities), as well as what the AQI of the city is. I am using the US AQI because that is used much more widely in the UK than the Chinese AQI.

```
10  struct AQI_Root_Layer0: Codable {
11      var data: AQI_Result_Layer1
12  }
13
14  struct AQI_Result_Layer1: Codable {
15      var city: String
16      var current: AQI_Result_Layer2
17  }
18
19  struct AQI_Result_Layer2: Codable {
20      var pollution: AQI_Result_Layer3
21  }
22
23  struct AQI_Result_Layer3: Codable {
24      var aqius: Int
25  }
```

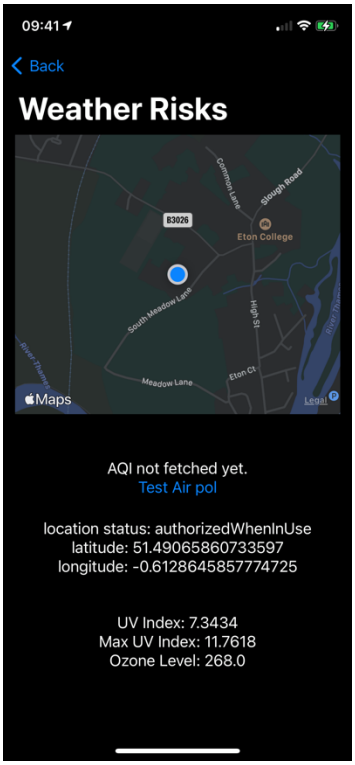
Then I created a button in the weather risks app to make a request for the co-ordinates 50.507, 0.1276 (which are the co-ordinates for London). At first the api request did not work, and I got the following error –

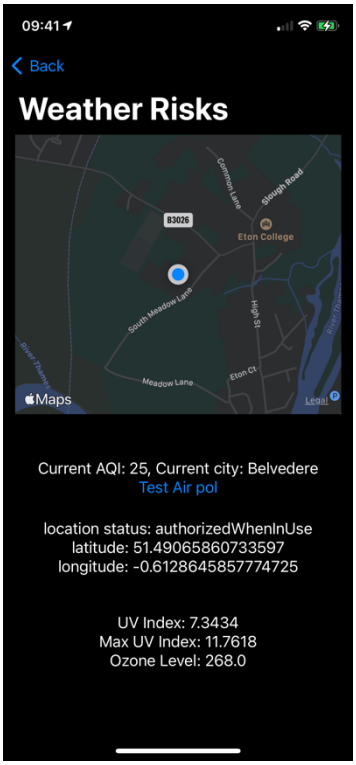
"The resource could not be loaded because the App Transport Security policy requires the use of a secure connection"

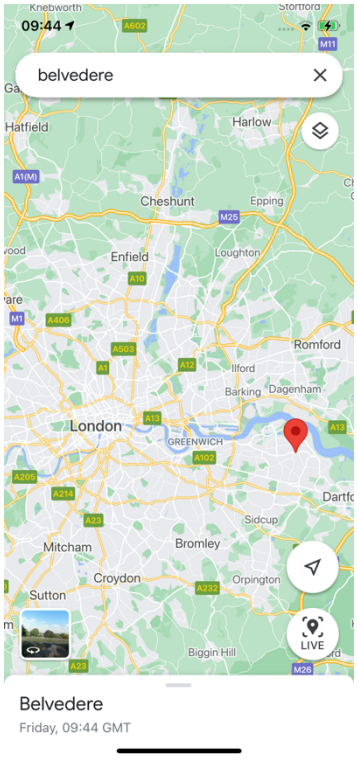
This meant I needed to add some extra details about the nature of the api request in the info.plist file. The information I had to add is shown below, as I could not do this using the interface in xcode, I had to open the raw info.plist file and add exceptions that allowed me to make this specific api request.

```
    <key>NSAppTransportSecurity</key>
    <dict>
        <key>NSAllowsArbitraryLoads</key>
        <true/>
        <key>NSExceptionDomains</key>
        <dict>

            <dict>
                <key>NSIncludesSubdomains</key>
                <true/>
                <key>NSThirdPartyExceptionRequiresForwardSecrecy</key>
                <false/>
            </dict>
        </dict>
    </dict>
```

The user interface after implementing the button and making sure the api request works is shown below. The location of the city is also shown to verify that the api request is indeed being made to the correct location (closest high accuracy UV light measuring station).







i. Ticket 8 – Retrieve air pollution level for location

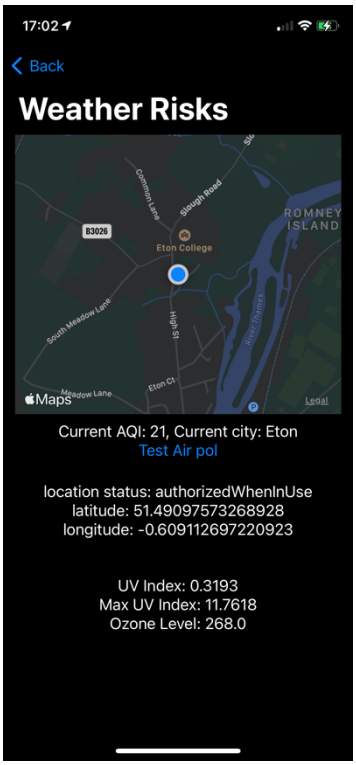
Next, I decided to make the api request work for the user's correct location. I did this by first updating the function in the AQIManager class so that it takes in a set of co-ordinates and returns the AQI for that location. This only needed me to update the url for the request and is shown below.

```
74      //same function but for the specific co-ordinates passed in
75
76      func requestAQIInfoForLocation(inputLatitude: Double, inputLongitude: Double, callback: @escaping (String, Int) -> ()) {
77          //defining what the request is
78          let request = NSMutableURLRequest(url: NSURL(string: "http://api.airvisual.com/v2/nearest_city?lat=" + String(inputLatitude) +
                "&lon=" + String(inputLongitude) + "&key=" + String(accessKey) )! as URL, cachePolicy: .useProtocolCachePolicy,
                timeoutInterval: 10.0)
79
```

Then I updated the button so that it passed in the user's current location when making the request.

```
45                  Text(AQIDataCheck)
46                  Button("Test Air pol") {
47                      aqiManager.requestAQIInfoForLocation(inputLatitude: Double(userLatitude) ?? 0, inputLongitude: Double(userLongitude) ?? 0,
                            callback: {(cityResponse: String, aqiResponse: Int) -> () in
48                          AQIDataCheck = "Current AQI: " + String(aqiResponse) + ", Current city: " + String(cityResponse)
49                      })
50                  }
```

The screenshot below shows how the city now changes to my current location, indicating that the api request is working accurately.

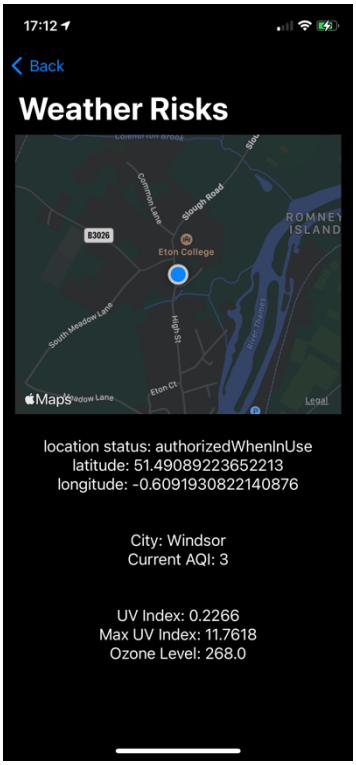


To make using the app a better experience for the user I made all of the api requests now run as soon as the user opens the reminders page, rather than the user having to press a button to get the information. I did this by using the onAppear functionality of views in xcode, shown below.

```
68          .navigationBarTitle("Weather Risks")
69          .onAppear(perform: {
70              uvIndexManager.requestUVInfoForLocation(inputLatitude: Double(userLatitude) ?? 0, inputLongitude: Double(userLongitude) ?? 0
                    ,callback: {(uvResponse: Double, uvMaxResponse: Double, ozoneResponse: Double) -> () in
71                  uvIndexDisplay = String(uvResponse)
72                  uvMaxIndexDisplay = String(uvMaxResponse)
73                  ozoneLevelDisplay = String(ozoneResponse)
74              })
75
76              aqiManager.requestAQIInfoForLocation(inputLatitude: Double(userLatitude) ?? 0, inputLongitude: Double(userLongitude) ?? 0,
                    callback: {(cityResponse: String, aqiResponse: Int) -> () in
77                  AQICity = String(cityResponse)
78                  AQIData = String(aqiResponse)
79              })
80
81          })
```

The final look of the UI now is shown below. I also realised a minor error in my previous testing whilst implementing this final bit of code. Previously whilst testing, I was using an emulated location in xcode, rather than the location from the phone used for testing. This caused me to get the AQI data for

London, whilst displaying the city in which my phone is. I fixed this by disconnecting the phone whilst testing the location feature.



j. Sprint Review

In this sprint I implemented apis to allow the user to see their UV light and pollution risks in their area. A few small problems were found when testing this feature post-sprint. Firstly, the map would sometimes not center on the user after the user tries to move the map. After some research I found that this was a bug with MapKit that I unfortunately cannot fix at the moment. As well as this, the apis often would time-out and not return anything when the user was using mobile data. I attempted to fix this by returning a message to the user to get a better signal when using mobile data to check the weather risks.

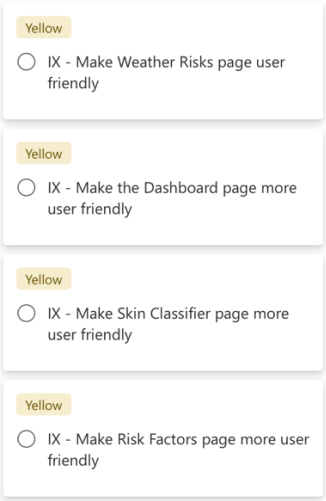
This sprint has worked on the following points specified in the design section:

- [1.a.8] UV light risk and pollution risk in area

10. Sprint 9 – Aesthetics and Final Testing

a. Ticket overview

These tickets are mostly for the user's quality of life. Almost all the functional aspect of the app is now finished, and I will spend the time I have remaining making the app more usable.



b. Ticket 1 – Make Weather Risks page user friendly

I wanted to make the app more visually pleasing, and so easier for the user to use. Therefore, I decided to make a colour-based visual that can aid users in judging whether the air quality level and the ultraviolet light level is good or bad. To stay consistent with the rest of the world's standards I used the following rules for my visual aspects –

UV Level Name	UV Level Short Name	UV Range	UV Colour
Low	Low	`[0-3)`	`#558B2F`
Moderate	Mod	`[3-6)`	`#F9A825`
High	High	`[6-8)`	`#EF6C00`
Very High	VHigh	`[8-11)`	`#B71C1C`
Extreme	Extr	`[11+...)`	`#6A1B9A`

(https://www.openuv.io/uvindex)

Daily AQI Color	Levels of Concern	Values of Index	Description of Air Quality
Green	Good	0 to 50	Air quality is satisfactory, and air pollution poses little or no risk.
Yellow	Moderate	51 to 100	Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sensitive to air pollution.
Orange	Unhealthy for Sensitive Groups	101 to 150	Members of sensitive groups may experience health effects. The general public is less likely to be affected.
Red	Unhealthy	151 to 200	Some members of the general public may experience health effects; members of sensitive groups may experience more serious health effects.
Purple	Very Unhealthy	201 to 300	Health alert: The risk of health effects is increased for everyone.
Maroon	Hazardous	301 and higher	Health warning of emergency conditions: everyone is more likely to be affected.

(https://www.airnow.gov/aqi/aqi-basics/)

Then I started off by removing all unnecessary parts of the user interface. I removed the longitude and latitude displayed (as that was useless to the user and was only needed for development).

For the visual display, I decided to use different coloured rounded rectangles. Depending on the air pollution or uv light levels a different rectangle would be more opaque than others. In order to implement this, I created a new file that would contain functions to return different views depending on the values passed in. The function for the UV rectangles is shown below.

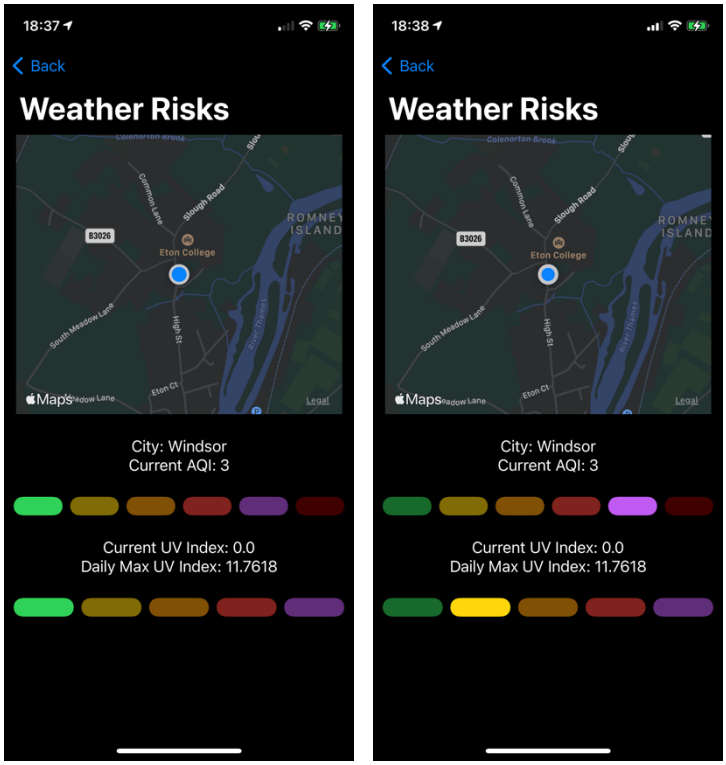
```swift
class WeatherRisksViewModel {

    //source - https://www.openuv.io/uvindex

    @ViewBuilder func getUVColourCodedRectangles(uv: Double) -> some View {

        if uv == -1 {

            HStack {
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.green)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.yellow)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.orange)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.red)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.purple)
            }.frame(width: UIScreen.screenWidth - 20, height: 20)

        } else if uv < 3 {

            HStack {
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.green)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.yellow)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.orange)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.red)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.purple)
                    .opacity(0.5)
            }.frame(width: UIScreen.screenWidth - 20, height: 20)

        } else if uv < 6 {

            HStack {
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.green)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.yellow)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.orange)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.red)
                    .opacity(0.5)
                RoundedRectangle(cornerRadius: 25, style: .continuous)
                    .fill(Color.purple)
                    .opacity(0.5)
            }.frame(width: UIScreen.screenWidth - 20, height: 20)

        } else if uv < 8 {

            HStack {
```

(Around 350 lines of if statements)

I then implemented this for the UV light as well (in a similar method). I also happened to make an extension during this, as I wanted to make an easy way to get the screen height and width. This would help me to make a user interface that would work on any device, regardless of how big the screen is. The extension's code is shown below.

```swift
import Foundation
import SwiftUI

//extension to get screen width and height for whatever device

extension UIScreen{
   static let screenWidth = UIScreen.main.bounds.size.width
   static let screenHeight = UIScreen.main.bounds.size.height
   static let screenSize = UIScreen.main.bounds.size
}

```

The UI at this stage is shown below. On the left side you can see what the UI looks like for data for my actual location, on the right I have used set AQI and UV values to demonstrate what would be shown if the levels of pollution/uv light were higher (and also for testing purposes).



At this point I realised that the "Daily Max UV Index" value would make no sense unless the user was given more data. Therefore, I added an additional part to my UV light api request, now also storing the "solarNoon" value (when the sun is at the peak during the day). This would provide information about when the max UV Index occurs. I did this by first expanding the data structure I used to store information from the json response to what is shown below.

```
11 struct UV_Root_Layer0: Codable {
12     var result: UV_Result_Layer1
13 }
14
15 struct UV_Result_Layer1: Codable {
16     var uv: Double
17     var uv_max: Double
18     var ozone: Double
19     var sun_info: UV_Result_Layer2
20 }
21
22 struct UV_Result_Layer2: Codable {
23     var sun_times: UV_Result_Layer3
24 }
25
26 struct UV_Result_Layer3: Codable {
27     var solarNoon: String
28 }
```

Furthermore, I made the api request function return the solarNoon value as well. Unfortunately, openUV claims that it returns the date in the iso6801 format, but in reality, it uses a custom format. This caused quite a few errors as I was attempting to decode the json using Apple's built in iso6801 date formatter. I had to create my own custom iso6801 converter, which is shown below.

```swift
class DateFormattingModel {

//    let inputDate = "2020-08-22T18:55:36Z"
//    let dateFormatter = DateFormatter()
//    dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
//    let date = dateFormatter.date(from: inputDate) ?? Date()

    func convertFromISO6801ToDate(inputISODate: String) -> Date {

        //2021-12-11T04:11:52.613Z example input

        var newTimeString = ""
        var startAdding = true

        //this produces a string until the input stops obeying iso6801 format
        for c in inputISODate {

            if c == "." {
                startAdding = false
            }

            if startAdding == true {
                newTimeString.append(c)
            }

        }

        //adding the end character
        newTimeString.append("Z")

        //formatting the date
        let inputDate2 = newTimeString.replacingOccurrences(of: "T", with: " ")
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd HH:mm:ssZ"
        let date = dateFormatter.date(from: inputDate2) ?? Date()
//        print(date)
//        print(newTimeString)

        return date
    }

}
```

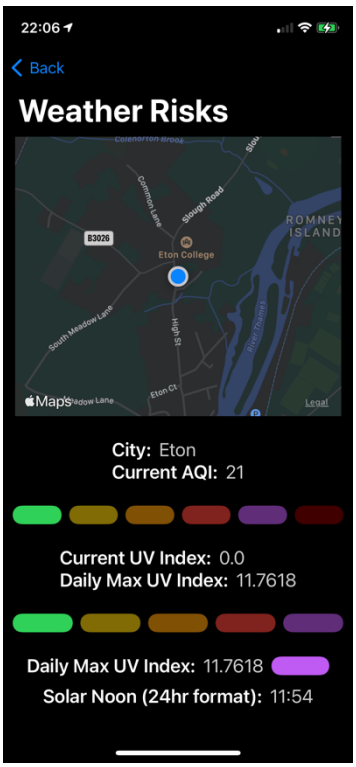
I used this to deformat the input iso6801-like date and display it (as shown in the code below).

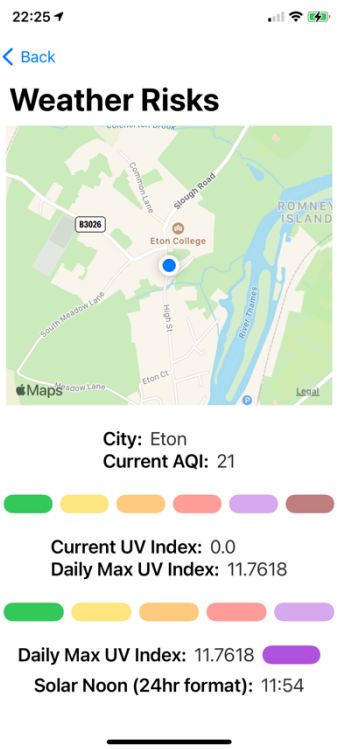
```swift
HStack {
    Text("Solar Noon (24hr format):")
        .font(.title3)
        .bold()
    //Text(solarNoonDisplay, style: .time)
    Text(dateFormattingModel.convertFromISO6801ToDate(inputISODate: solarNoonDisplay), style: .time)
        .font(.title3)
        .opacity(0.8)
}
```

I also restructured a lot of the displayed items in the view so that they were more readable and stress was made on certain aspects of the view to make it more usable. The change in some of the code is shown below, most of the items in the view followed a similar structure.

```
                VStack(alignment: .leading) {

                    HStack {
                        Text("City:")
                            .font(.title3)
                            .bold()
                        Text("\(AQICity)")
                            .font(.title3)
                            .opacity(0.8)
                    }

                    HStack {
                        Text("Current AQI:")
                            .font(.title3)
                            .bold()
                        Text("\(AQIData)")
                            .font(.title3)
                            .opacity(0.8)
                    }

                }.padding()

                weatherRisksViewModel.getAQIColourCodedRectangles(aqi: Double(AQIData) ?? -1)

                //uncommment to test for different AQI values
                //weatherRisksViewModel.getAQIColourCodedRectangles(aqi: 260)

                VStack(alignment: .leading) {
                    HStack {
                        Text("Current UV Index:")
                            .font(.title3)
                            .bold()
                        Text("\(uvIndexDisplay)")
                            .font(.title3)
                            .opacity(0.8)
                    }

                    HStack {
                        Text("Daily Max UV Index:")
                            .font(.title3)
                            .bold()
                        Text("\(uvMaxIndexDisplay)")
                            .font(.title3)
                            .opacity(0.8)
                    }
                }.padding()

                weatherRisksViewModel.getUVColourCodedRectangles(uv: Double(uvIndexDisplay) ?? -1)
```

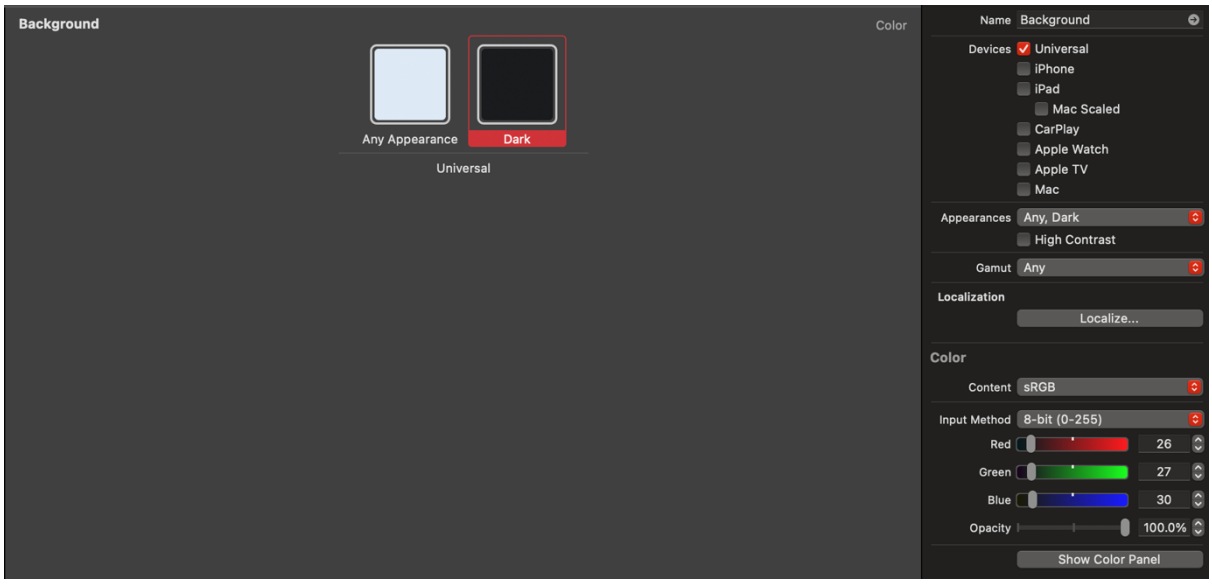
The final user interface is shown below, both in Apple's light and dark mode.





c. Ticket 2 – Make the Dashboard page user friendly

I started off by creating a colour set for my app, as the generic black and white did not look great as the colour scheme for the app. I created colour elements in the assets.xcassets file in xcode. In that file, I created two colours – one for the light mode version of the app, and another for the dark mode version of the app.



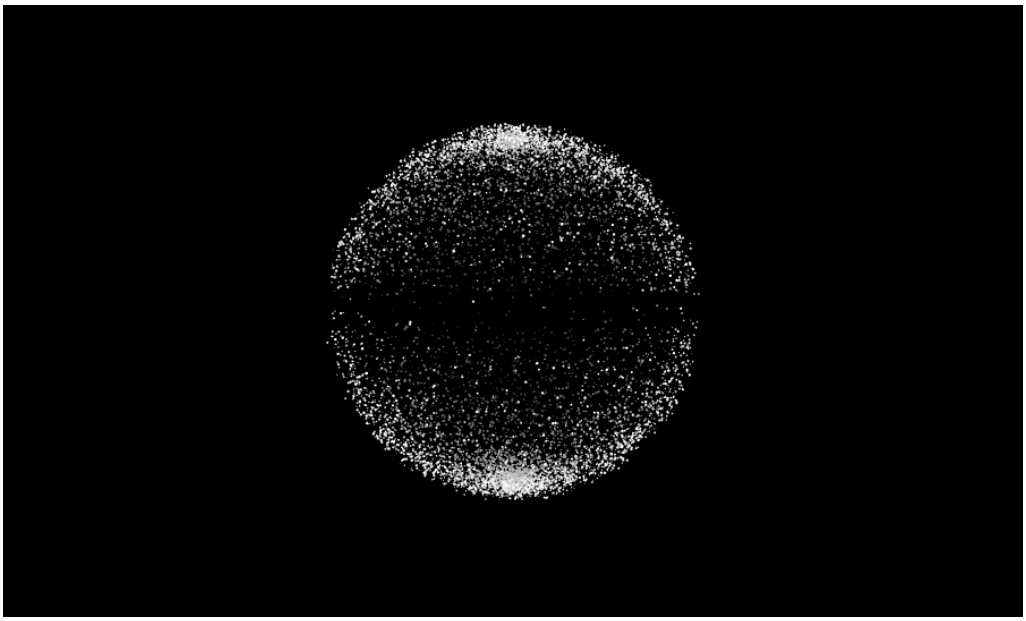
I used these custom light/dark mode colours by create a vertical stack of elements in each of my views, with the background being an infinite rectangle of the custom colour. The code for this is shown below.

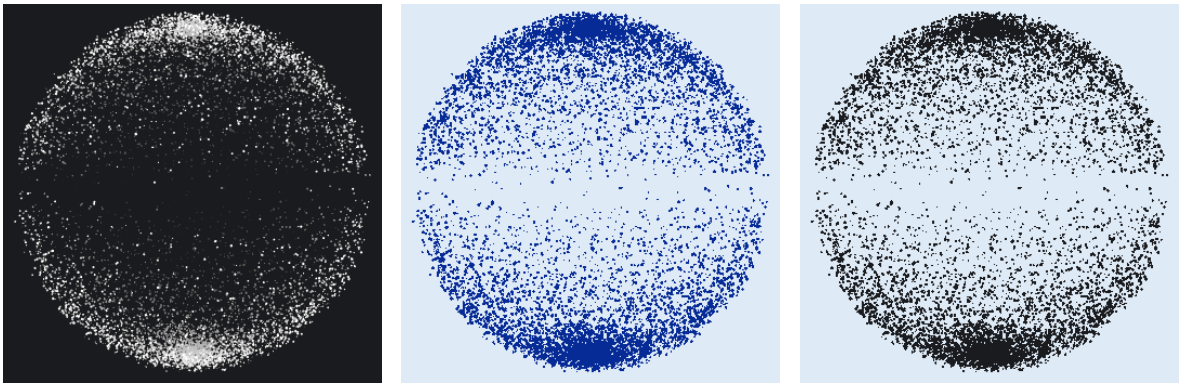
```
        ZStack {
            Rectangle()
                .fill(Color("Background"))
                .frame(maxWidth: .infinity, maxHeight: .infinity)
                .edgesIgnoringSafeArea(.all)

```

Then I started to work on creating buttons that would be visually pleasing, and easier to use than clicking on some text. I settled on creating minimalistic rounded rectangle buttons with slight shadows behind them. I created a “ButtonStyle” to achieve this, and the detailed methods I used to create these buttons are shown below.

```
12  struct NeumorphicButtonStyle: ButtonStyle {
13      var color: Color
14
15      //creates a button that has slight shadows and looks nice
16      func makeBody(configuration: Self.Configuration) -> some View {
17          configuration.label
18              //.padding()
19              //.background(RoundedRectangle(cornerRadius: 20, style:
                    .continuous).fill(Color("Background")).frame(width: UIScreen.screenWidth - 40, height:
                    UIScreen.screenHeight/15).shadow(color: Color("LightShadow"), radius: 8, x: -8, y:
                    -8).shadow(color: Color("DarkShadow"), radius: 8, x: 8, y: 8) )
20              //.frame(width: UIScreen.screenWidth/1.11, height: UIScreen.screenHeight/15)
21              .frame(width: UIScreen.screenWidth/3, height: UIScreen.screenHeight/50)
22              .padding(.horizontal, (UIScreen.screenWidth/4))
23              .padding(.vertical, (UIScreen.screenHeight/40))
24              //.padding(10)
25              .background(Color("Background"))
26              .cornerRadius(20)
27              .shadow(color: Color("DarkShadow"), radius: 20, x: 8, y: 8)
28              .shadow(color: Color("LightShadow"), radius: 20, x: -8, y: -8)
29              .scaleEffect(configuration.isPressed ? 0.9 : 1.0)
30      }
31
32      // for reference -
33      //.background(RoundedRectangle(cornerRadius: 20, style:
            .continuous).fill(Color("Background")).frame(width: UIScreen.screenWidth - 40, height:
            UIScreen.screenHeight/15).shadow(color: Color("LightShadow"), radius: 8, x: -8, y:
            -8).shadow(color:  Color("DarkShadow"), radius: 8, x: 8, y: 8) )
34
35  //    Button("Test Button22222"){
36  //    }.frame(width: UIScreen.screenWidth - 40, height: UIScreen.screenHeight/15)
37  //        .buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
38  //        .padding()
39  }
```

Having made aesthetically pleasing buttons, the dashboard still lacked anything unique that made it a nice view. Therefore, I decided to add a gif to the view, which would act as the button leading to the melanoma scanner (the main part of the app, which is why it should get extra attention). The gif I decided to use was of a 3d representation of spherically orbiting particles, which seemed to fit the app’s usage of AI. Below I have shown the gif I used, and a few of the many iterations I went through editing the images to make them usable with my colour scheme.





Unfortunately, swiftUI does not officially support gifs (there is no inbuilt function to deal with gifs), therefore when I tried to add the gif image to the UI, it only displayed a static image. I attempted many ways of getting around this problem, such as creating UIImage extensions that looped through images one at a time. Unfortunately, that method was extremely memory inefficient, so I decided to start again using a different method.

I ended up using apple's WebKit service to create a webview embedded in a swiftUI view to show the image. Then I disabled any clicking/scrolling on the webview so the users will not know that it is a web view that they are being shown. In this way I can display a gif image through an embedded http view (essentially a website being displayed inside the app). The code I used to embed the gif image is shown below.

```swift
import Foundation
import WebKit
import SwiftUI
import UIKit


struct GifImage: UIViewRepresentable {
    private let name: String

    init(_ name: String) {
        self.name = name
    }

    //creating the actual webview using the gif file added to the files.
    func makeUIView(context: Context) -> WKWebView {
        let webView = WKWebView()
        let url = Bundle.main.url(forResource: name, withExtension: "gif")!
        let data = try! Data(contentsOf: url)

        webView.load(
            data,
            mimeType: "image/gif",
            characterEncodingName: "UTF-8",
            baseURL: url.deletingLastPathComponent()
        )

        //webView.scrollView.isScrollEnabled = false
        //webView.scrollView.contentInset = UIEdgeInsets.zero
        //webView.scrollView.contentInset = UIEdgeInsets(top: 0, left: 0, bottom: 0, right: 0)
        webView.scrollView.isUserInteractionEnabled = false
        webView.scrollView.backgroundColor = UIColor(Color("Background"))

        return webView
    }

    //updating the webview (making sure the gif animates)
    func updateUIView(_ uiView: WKWebView, context: Context) {
        uiView.scrollView.contentInset = UIEdgeInsets(top: 0, left: 0, bottom: 0, right: 0)
        uiView.reload()
    }

}

```

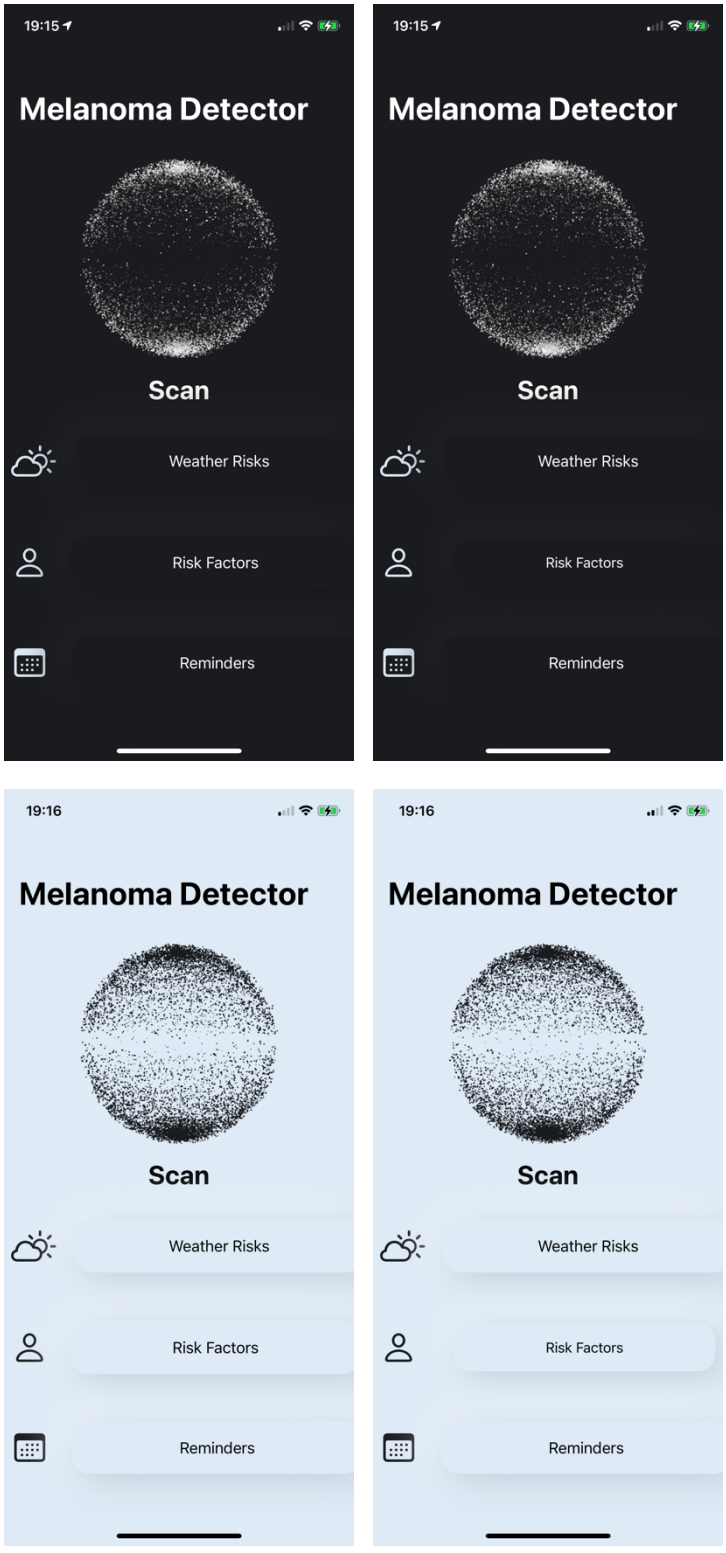
I added the gif to my dashboard view using the following code –

```
11
12      @Environment(\.colorScheme) var colorScheme
13      @State var scanImage = GifImage("Light_Mode_Cropped")
14
15      var body: some View {
16
17          // a navigation view allows for links to other pages
18          NavigationView {
19
20              ZStack {
21                  Rectangle()
22                      .fill(Color("Background"))
23                      .frame(maxWidth: .infinity, maxHeight: .infinity)
24                      .edgesIgnoringSafeArea(.all)
25
26                  VStack {
27
28
29                      NavigationLink(destination: SkinClassifier()) {
30                          VStack {
31                              if colorScheme == .dark {
32                                  GifImage("Dark_Mode_Cropped")
33                              } else {
34                                  GifImage("Light_Mode_Cropped")
35                              }
36                          }
37                      }.frame(width: 225, height: 225)
38                          .background(Color.clear)
39
40                      if colorScheme == .dark {
41                          Text("Scan")
42                              .font(.title)
43                              .bold()
44                              .foregroundColor(.white)
45                              .opacity(1)
46                      } else {
47                          Text("Scan")
48                              .font(.title)
49                              .bold()
50                              .foregroundColor(.black)
51                              .opacity(1)
52                      }
53
```

As well as this I decided to use apple's SFSymbols to add some more visual aspects to the buttons. Next to each of the buttons, I added symbols that were related to the view the button led to and made sure that they were coloured using the new colour format. The code for one of the buttons is shown below, each of them follow a near identical format (except for where the link leads to and parameters passed into the view).

```
66                      HStack {
67                          Image(systemName: "person")
68                              .font(Font.system(.largeTitle))
69                              .foregroundColor(Color("Anti-Background"))
70                              .padding()
71                          NavigationLink("Risk Factors", destination: InformationPage(coreDataManager: CoreDataManager()))
72                              .buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
73                      }.padding()
74
```

The UI after this ticket is shown below. Unfortunately, I cannot show the gif in this document, but the visual at the top does rotate.



c. Ticket 3 – Make Skin Classifier page user friendly

I started off by making all the buttons in this page consistent with the rest of the app, changing their button style to the custom one I made earlier. I also changed the text in all the buttons to make them more concise.

```
82                  //button to call subroutine to classify image
83                  Button("Classify Image") {
84                      showingClassificationWarning = true
85                      (self.classificationLabel, self.confidence) =
                            imageClassifierInstance.performImageClassification2(image:
                            imageSelectedFromCameraRoll)
86                      //converts confidence from 0-1 to 0-100, adding normalisation
87                      self.confidence = imageClassifierInstance.certaintyFunction(oldCertainty:
                            self.confidence)
88                  }.buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
```

Furthermore, I made the output information easier to read, and more prominent in the view.

```
107             //what the image is classified as
108             Text(classificationLabel)
109                 .bold()
110                 .foregroundColor(Color("Background"))
111                 .padding()
112                 .background(RoundedRectangle(cornerRadius: 20).foregroundColor(Color("Anti-Background")))
113                 .padding()
114                 .font(.title)
115
116             //confidence of classification
117             Text("Classifier Confidence: " + String(confidence))
118                 .padding()
119                 .font(.title3)
```

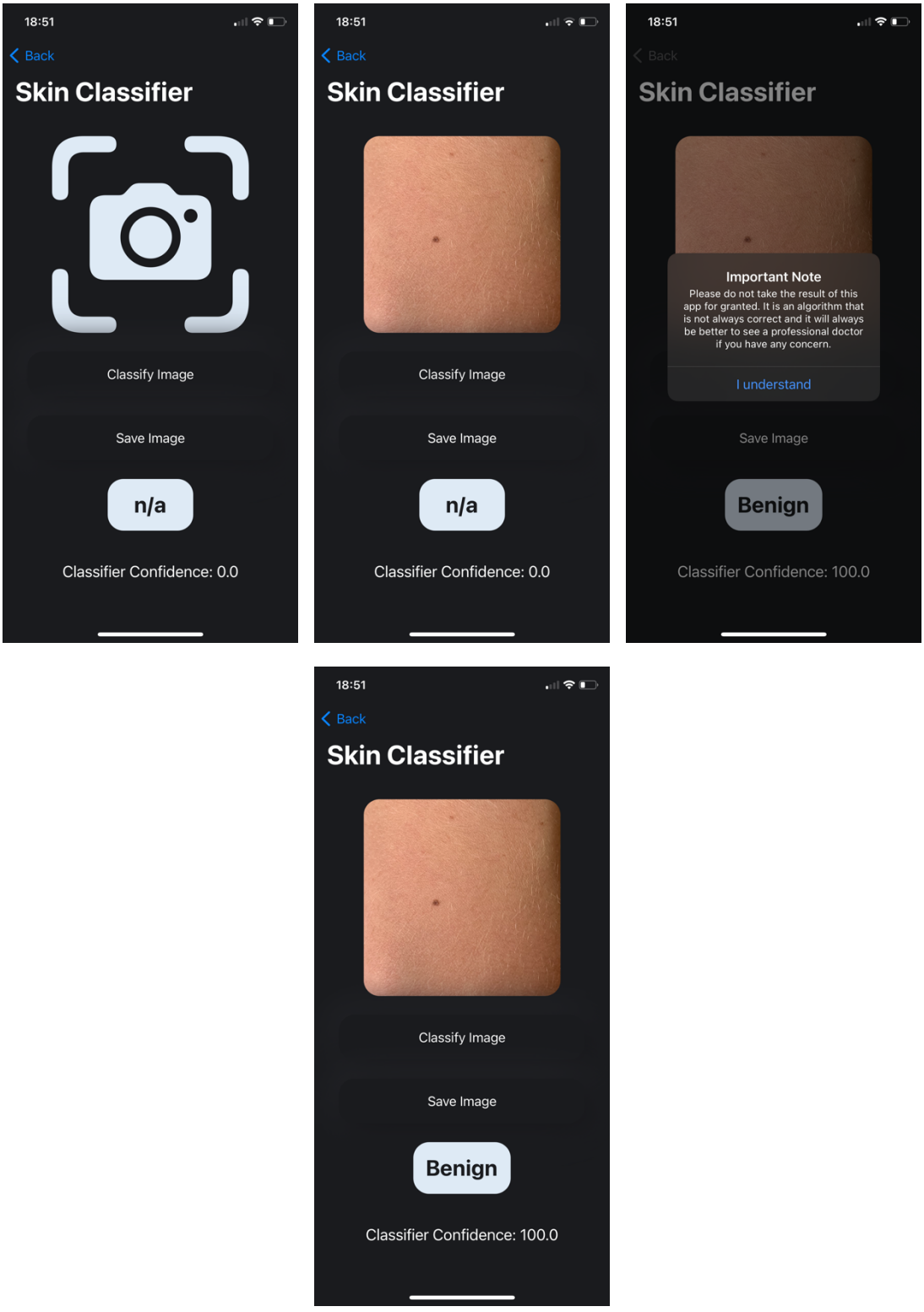
Then I added the correct information to the alert shown when you click on “classify image” in the app (which was previously full of placeholder information).

```
83                  Button("Classify Image") {
84                      showingClassificationWarning = true
85                      (self.classificationLabel, self.confidence) =
                            imageClassifierInstance.performImageClassification2(image:
                            imageSelectedFromCameraRoll)
86                      //converts confidence from 0-1 to 0-100, adding normalisation
87                      self.confidence = imageClassifierInstance.certaintyFunction(oldCertainty:
                            self.confidence)
88                  }.buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
89                  .alert( isPresented: $showingClassificationWarning) {
90                      Alert(
91                          title: Text("Important Note"),
92                          message: Text("Please do not take the result of this app for granted. It is an
                                algorithm that is not always correct and it will always be better to see a
                                professional doctor if you have any concern."),
93                          dismissButton: .default(Text("I understand"), action: {
94                          })
95                      )
96                  }.padding()
```

I also made all the corners of every element in the app rounded, so that they are more user friendly and fit the aesthetic of the whole app. The placeholder image for the skin classifier was also replaced with an icon from SF Symbols.

```
47                  Button(action: {
48
49                      showChooseCameraOrRollSheet = true
50
51                  }, label: {
52                      if changeClassificationImage == true {
53                          Image(uiImage: imageSelectedFromCameraRoll)
54                              .resizable()
55                              .cornerRadius(20)
56                              .frame(width: 250, height: 250)
57                      } else {
58                          Image(systemName: "camera.viewfinder")
59                              .resizable()
60                              .cornerRadius(20)
61                              .foregroundColor(Color("Anti-Background"))
62                              .frame(width:250,height:250)
63                      }
64                  }).actionSheet(isPresented: $showChooseCameraOrRollSheet) {
65                      ActionSheet(title: Text("Select Photo"),
```

The UI of the skin classifier view after this ticket is shown below.



c. Ticket 4 – Make Risk Factors page user friendly

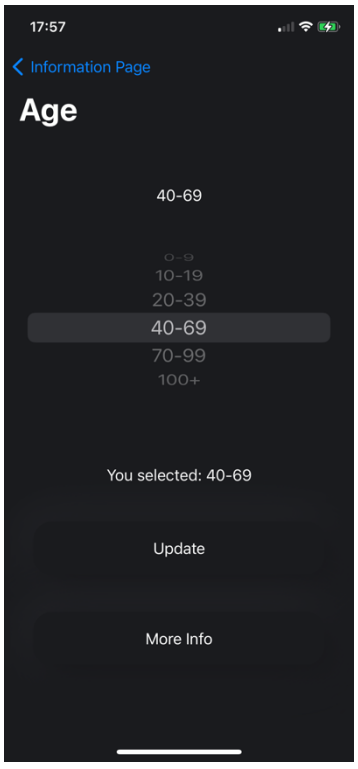
To make the risk factors page more usable for everyone I added a "more information" button onto each page for the risk factors. This button opens a link in the user's default web browser to cancer research uk's information about that specific risk factor. The website is also where I got a lot of my information when developing the app (link: https://www.cancerresearchuk.org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors).

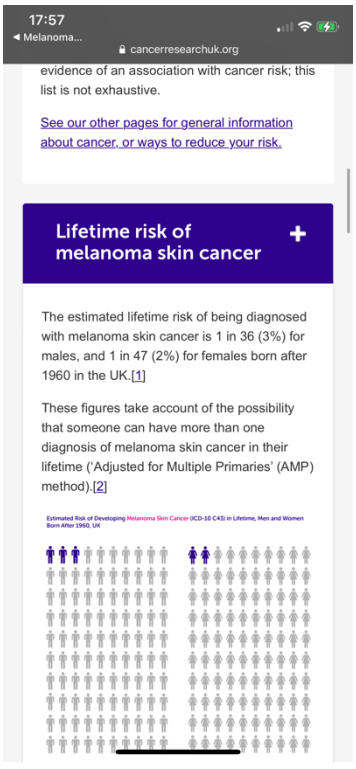
I implemented this by adding a button that opens the browser with the link of the correct risk factor the user is currently looking at. The code and UI is shown below.

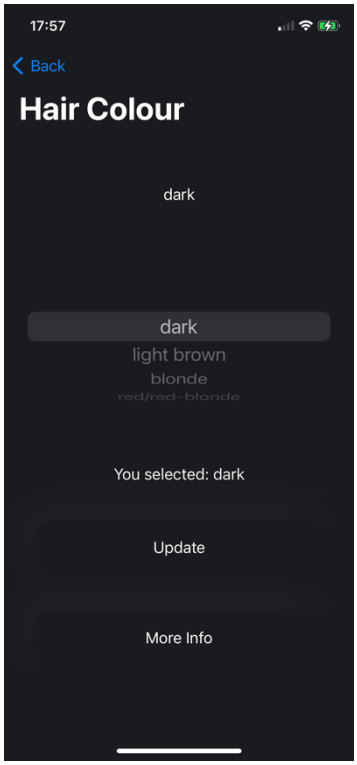
I also removed the dev value from being shown in the app.

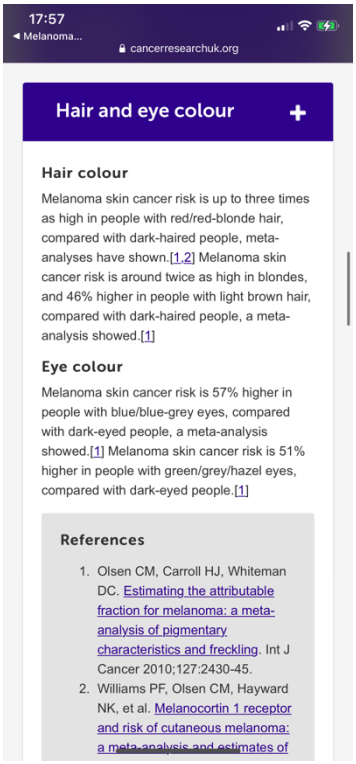
```
57
58                  //link to risk factors in cancer research uk website.
59                  Link("More Info", destination: URL(string: riskFactorsModel.returnMoreInfoURL(name: riskFactor.name ??
                        "https://www.cancerresearchuk
                        .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors"))!)
                        .buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
60                      .padding()
61              }.navigationBarTitle(riskFactor.name ?? "")
```

```
184     func returnMoreInfoURL(name: String) -> String {
185         if name == "Age"{
186
187             return
                    "https://www.cancerresearchuk
                    .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors#collapseZ
                    ero"
188
189         } else if name == "Gender" {
190
191             return
                    "https://www.cancerresearchuk
                    .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors#collapseZ
                    ero"
192
193         } else if name == "Skin Type" {
194
195             return
                    "https://www.cancerresearchuk
                    .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors#collapseF
                    our"
196
197         } else if name == "Eye Colour" {
198
199             return
                    "https://www.cancerresearchuk
                    .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors#collapseF
                    ive"
200
201         } else if name == "Hair Colour" {
202
203             return
                    "https://www.cancerresearchuk
                    .org/health-professional/cancer-statistics/statistics-by-cancer-type/melanoma-skin-cancer/risk-factors#collapseF
                    ive"
204
205         } else if name == "Number of Moles" {
```



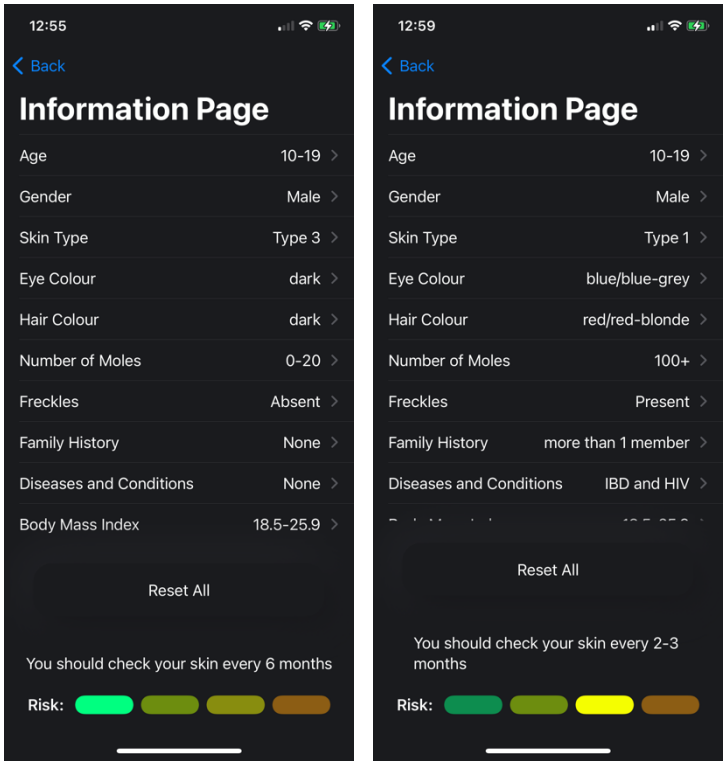






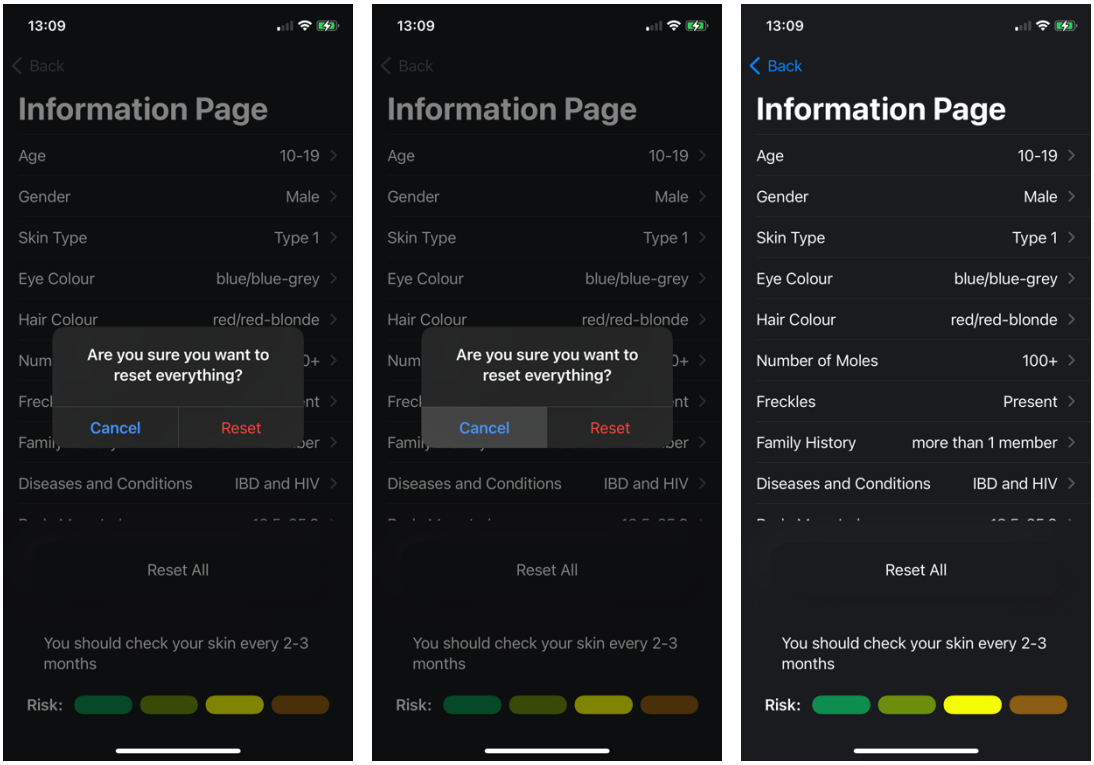
Next, I decided to make the risk factor value that the user sees more user friendly. I did this by replacing the risk factor raw value with a visual representation that I developed for the weather risks section (with the colours changed). And then to allow the user to have a concrete use for the risk factor I gave a recommendation for how often they should be checking their skin for melanoma. The code and the final UI are shown below.

```swift
class InformationPageViewModel {

    @ViewBuilder func getRiskFactorColourCodedRectangles(riskFactorVal: Double) -> some View {

        if riskFactorVal == -1 {

            VStack {
                Text("n/a").padding()
                HStack {
                    Text("Risk: ").bold()
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 0 / 255, green: 255 / 255, blue: 128 / 255)).opacity(0.5)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 192 / 255, green: 255 / 255, blue: 0 / 255)).opacity(0.5)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 245 / 255, green: 255 / 255, blue: 0 / 255)).opacity(0.5)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color.orange.opacity(0.5))
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                }
            }.padding(.bottom)

        } else if riskFactorVal <= 2 {

            VStack {
                Text("You should check your skin every 6 months").padding()
                HStack {
                    Text("Risk: ").bold()
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 0 / 255, green: 255 / 255, blue: 128 / 255)).opacity(1)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 192 / 255, green: 255 / 255, blue: 0 / 255)).opacity(0.5)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color(red: 245 / 255, green: 255 / 255, blue: 0 / 255)).opacity(0.5)
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                    RoundedRectangle(cornerRadius: 25, style: .continuous)
                        .fill(Color.orange.opacity(0.5))
                        .frame(width: UIScreen.screenWidth/6, height: 20)
                }
            }.padding(.bottom)

        } else if riskFactorVal <= 3.5 {


                //Text( riskFactorsModel.calculateFinalRiskFactor(riskFactorsList: riskFactorsList)
                    ).padding()
                informationPageViewModel.getRiskFactorColourCodedRectangles(riskFactorVal:
                    Double(riskFactorsModel.calculateFinalRiskFactor(riskFactorsList: riskFactorsList))
                    ?? -1 )


```



I also decided to implement a feature that I was told to by a stakeholder – a confirmation button for the reset button. The code and the associated functionality are shown below.

```
54              Button("Reset All") {
55                  showingResetRiskFactorsAlert = true
56              }.buttonStyle(NeumorphicButtonStyle(color: Color("Background")))
57                  .padding()
58                  .alert(isPresented: $showingResetRiskFactorsAlert) {
59                      Alert(
60                          title: Text("Are you sure you want to reset everything?"),
61                          message: Text(""),
62                          primaryButton: .destructive(Text("Reset")) {
63                              riskFactorsModel.resetRiskFactors(coreDataManager: coreDataManager,
                                    listOfRiskFactors: stringListOfRiskFactors)
64                              populateRiskFactors()
65                          },
66                          secondaryButton: .cancel()
67                      )
68                  }
69
70              //text showing numerical risk factor value
71              //Text( riskFactorsModel.calculateFinalRiskFactor(riskFactorsList: riskFactorsList) ).padding()
72              informationPageViewModel.getRiskFactorColourCodedRectangles(riskFactorVal:
                    Double(riskFactorsModel.calculateFinalRiskFactor(riskFactorsList: riskFactorsList)) ?? -1 )
73
74
```



d. Sprint Review

In this sprint I worked on making the app much more usable. I converted the app from just a functionality-based app to one that can be used by anyone in an enjoyable way. In the next section of the documentation, I will test the app as a whole and further improve upon it, removing any errors or problems that arise. Furthermore I will be implementing any significant stakeholder feedback.

This sprint has worked on the following points specified in the design section:

- General structure improvement