Skin Disease Classification App

Skinnefy

Naman Pujari – Siddharth Rajan – Harsh Patel – Faheem Kamal

CSC 59867 Senior Design Project – Spring 2020

Professor George Wolberg

TABLE OF CONTENTS

ABSTRACT

Over the past two semesters, an iOS/Android compatible application capable of diagnosing skin conditions was developed using React Native, and a Convolution Neural Network hosted on Amazon Web Services (AWS). Datasets were assembled using various sources, including the likes of Dermnet – an archive of skin condition related images – and Kaggle a web hosted library of Machine Learning datasets. Using SageMaker, multiple image classification models were developed, ranging from ones that identify 3 classes to ones that identify 9. Throughout each iteration, the training and validation accuracies were increased to satisfying values. Using the AWS JavaScript SDK (software development kit), the React Native application was able to communicate with the SageMaker-hosted custom skin condition classification model, and enabled predictions based on real-time images, obtained from users' smartphone cameras.

Throughout the project, multiple technologies were considered, and directions were changed due to various reasons. Technologies that were not moved further with had shortcomings when considered with the scope of the project in mind, a lot of which will also be discussed in this report.

INTRODUCTION

a. Background

Skin conditions affect an estimate of 1.9 billion people and is the fourth leading cause of a non-fatal burden worldwide. Due to a shortage of dermatologists and wait times, access to dermatology care is limited. Furthermore, diagnostic accuracy of general practitioners is 0.24 to 0.70 compared to 0.77 to 0.96 for dermatologists, which result in over-referrals, under-referrals, delay in proper treatment and significant errors in diagnosis. Skin conditions also affect 30 to 70% of individuals and is prevalent in all regions and age groups throughout the world. And because of the limited short supply of dermatologists, the burden of diagnosis falls in the hands of nurse practitioners (NPs), primary care physicians (PCPs) and physician assistants.

However, in recent years, to expand access to specialists and improve accuracies of diagnosis, store-and-forward tele-dermatology has become popular. This allows consumers to have an appointment remotely. But also, the use of artificial intelligence, might be more promising to broaden the availability of the expertise provided by dermatologists. Recent developments in deep learning now allow for the development of tools to assist in diagnosing skin conditions from snapshots taken from mobile devices.

b. Motivation

The main motivation for this project stems from wanting a proper and convenient solution that is available to consumers worldwide on any iOS and Android devices. We wanted to develop an application that can diagnose skin conditions from snapshots and proceed to provide in-depth information on conditions for consumers using a Deep Learning system with React Native as the framework for our application. On top of that, we found that developing a

proper application is highly beneficial as current competitors either cannot provide accurate results or forces consumers to pay an annual fee to use their products.

c. Related Works

i. Aysa

Aysa is a consumer-facing dermatology mobile application that is easy-to-use for common skin conditions. It is powered by VisualDX, a clinical decision support tool maker, and quickly analyzes your photo to provide personalized guidance. It utilizes machine learning to identify skin conditions and make treatment suggestions. This is done by taking a snapshot of the affected skin using your mobile device and then proceeds, to ask a few questions to refine results. Afterwards, it shows you the results and advises you to take a course of action. Aysa is currently used by more than 2,300 hospitals and clinics around the world.

ii. FirstDerm AutoDerm

A new app called Skin Image Search by First Derm, launched recently over 2 years ago. This aimed to use artificial intelligence to help people diagnose their skin conditions. They drew inspiration from the fact that Google image searches are wrongly labeled, so they hoped that through the development of their application, they can give a better understanding of the skin conditions that consumers may potentially have. User can upload a photo of their skin to the platform and the app will match the image to possible conditions using AI. However, the app only served as a skin image search and had no functionalities to contact doctors or give in-depth information of the skin conditions the users of the Apps were experiencing.

iii. A deep learning system for differential diagnosis of skin disease

Researchers at Cornell University with the assistance of Google, MIT, Medical University of Graz and the University of California, San Francisco developed a deep learning system to identify 26 of the most common skin conditions in adult cases that were referred for tele dermatology consultation. Instead of using a single classification between a small number of conditions, their deep learning system provides a differential diagnosis across 26 conditions that include various dermatitides, dermatoses, pigmentary conditions, and lesions to aid with the clinical decisions. Their DLS not only relies on images but on demographic information and medical history. And they verified the accuracy through a set of tests conducted by different board-certified clinicians across the world at three different levels of trainings: dermatologists, PCPs and NPs.

Their DLS has two major components: a variable number of deep CNN modules that process their input images and a shallow module to process metadata such as the patient's demographic information and medical history. To develop and validate their DLS, they applied a temporal split to data from a teledermatology service: the first 80% of the past decade was used for developments while the last 20% of the past decade were used for validation. To avoid bias, no patients were present in the development and validation of the DLS. Each case in the development stage was reviewed by a rotating panel between multiple dermatologists to determine diagnosis while in the validation stage, only three US board-certified dermatologists.

Their research and development of their DLS proved to be useful in our work. We determined that their research would be a helpful guide in the development of our app Skinnefy.

THEORY

a. Image Classification

Image Classification is a complex process in computer vision that can classify an image according to its visual content. It is a process that takes an input (like a picture) and outputs a class or a probability that the input is of a particular class. The intent is to categorize all pixels in an image into one of the several classes. It is one of the most important parts of digital image analysis. Two main classification methods are Supervised classification and Unsupervised classification.

Unsupervised classification is where the outcomes are based on a software analysis of an image to determine which pixels are related and as a result, they are grouped together into classes. Users can specify which algorithm the software can use and the desired number of output classes. The user does not aid in the classification process in general which reduces analyst bias. However, the user must have knowledge of the area that is being classified and it does require the user to assign labels and combine classes after the fact into useful information classes.

Supervised classification is a technique used for quantitative analysis of sensing image data. The user specifies various pixel values or spectral signatures that are associated with each class. This is done when the user specifies training sites, representative sample sites of a known cover type, to be used for classification. The computer then assigns pixels to the closest class based on the training data using an algorithm and the spectral signatures from these training areas. In Supervised classification, most of the effort is done prior to the actual process of classification. Once the classification is to run the output is a thematic image that is labeled with classes and correspond to the information classes. Moreover, it is more accurate than Unsupervised classification, but heavily depends on the training sites, the skills of the individual that is processing that image and the spectral distinctness of the classes. It also requires more time and money compared to Unsupervised classification. However, Unsupervised classification would yield more error because the spectral classes would contain more mixed pixels than the Supervised approach.

b. Convolutional Neural Network

Neutral networks are computing systems that is based on biological neural networks found in the brains of animals. Such systems perform tasks without being programmed with task-specific rules. It contains a collection of nodes called artificial neurons, which is loosely models after the neurons found in brains. Each connection, which act like that of a synapse in a brain, transmit signals to other neurons.

A Convolution Neural Network or CNN for short, is a class of deep neural network. The name itself indicates that the network employs convolution, a mathematical operation. CNN are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. A CNN consists of an input and an output layer, along with multiple hidden

layers or multilayer perceptron. It is a feedforward neural network where units are organized into layers and units at a given layer only get input from units in the layer below. At each layer, units are organized into 2-D grids called feature maps. Each of these maps is a result of a convolution performed on the layer below. This generally means that the same set of weights is applied at each location in the layer below. And this means, a unit can only receive inputs from units at the same location from a layer below. Moreover, the weights are the same for each unit in a feature map. After convolution, a few other computations are done such as cross-feature normalization. Here, the activity of a unit at a spatial location is divided by the activity of units at the same location in other feature maps. A more common operation is pooling where the maximum activity in a small spatial area represents that area. As a result, this shrinks the size of the feature maps. Convolution neural networks have proven to be very effective in areas such as image recognition and classification. CNNs have been successful in identifying faces, objects, and traffic signs.

c. Deep Residual Learning for Image Recognition

Deeper neural networks are difficult to train. However, researchers at Microsoft worked on a project that presented a residual learning framework to ease the training of networks that are substantially deeper. They reformulated the layers as learning residual functions with reference to the layer inputs instead of using unreferenced functions. In their paper, they presented that residual networks are easier to optimize and can gain accuracy from considerable increased depth. Instead of hoping that each stacked layer would fit a desired underlying mapping, they explicitly stated that these layers fit a residual mapping. They also let the stacked layer fit another mapping and their original mapping was recast into another function. Furthermore, they hypothesized that is easier to optimize the residual mapping than to optimize the original, unreferenced mapping.

They presented comprehensive experiments on ImageNet to show their degradation problem and evaluated their method. They showed that deep residual nets are easy to optimize and that their counterpart “plain” nets exhibit higher training error when the depth increases. This suggested that their deep residual nets have higher accuracy gains from greatly increased depth, producing results substantially better than previous networks. This research provided inspiration in the development of our application.

IMPLEMENTATION

a. Machine Learning Model

i. Python Modules for Locally Hosted Approaches

The sub disciplines of Machine Learning and Artificial Intelligence, when first introduced to the mainstream industry had their tasks performed by manually coding all the algorithms and mathematical and statistical formula. Concepts such as gradient descent, learning rate, backtracking, backpropagation and the like had to be implemented manually in order to achieve optimal results. This made the process very time consuming, tedious, and inefficient. As

a result, in these modern days, we have access to accessible python libraries, frameworks, and modules that make it easier and efficient to perform mathematically loaded computations. With its vast collection of libraries, Python has become one of the most popular programming languages for Machine Learning tasks and projects. These libraries are often made by employees of companies leading research in these areas. Some of the most popular frameworks in use today for deploying projects include: TensorFlow, PyTorch, and Keras.

TensorFlow

TensorFlow is an open source library for numerical computation and large-scale machine learning, created by the Google Brain Team. TensorFlow allows developers to create *dataflow graphs*—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or *tensor*. Nodes and tensors in TensorFlow are Python objects, and TensorFlow applications are themselves Python applications.

Though TensorFlow is a Python library, it is important to note that the actual math operations that occur within the functions of the library, are written in high-performance C++. Python just directs traffic between the pieces and provides high-level programming abstractions to combine them.

TensorFlow applications can be run on most any target that's convenient: a local machine, a cluster in the cloud, iOS and Android devices, CPUs or GPUs. Once you have the training model complete, you can deploy it on almost any device where it will be used to make predictions.

The single biggest benefit TensorFlow provides for machine learning development is abstraction and its simplicity. Rather than deal with the complex theoretical details of implementing algorithms, developers can focus on the overall logic and design of the application. All the behind the scenes of handling the inputs and outputs of different functions are taken care of by TensorFlow. There is also the *eager execution* mode that lets you evaluate and modify each graph operation separately and analyze it independently. This feature makes it easier debug and see the application's run-down. One issue that arises with using TensorFlow is that some details of TensorFlow's implementation make it hard to obtain totally deterministic model-training results for some training jobs. This means that a model trained on one system will sometimes vary slightly from a model trained on another, even when they are fed the exact same data. The reasons behind this are up for debate, but it is possible to avoid these issues. The Google Brain team is also considering adding more controls to minimize these inconsistencies.

With the backing and support of Google, an industry giant in the field of research and innovation, TensorFlow has become one of the most popular Python libraries for Machine Learning research. Thanks frameworks that ease the process of acquiring data, training models, serving predictions, and refining future results, developers can focus on the actual design of the application instead of worrying about the intricate aspects of the mathematical algorithms

PyTorch

PyTorch is an open source Python-based scientific computing package that uses the power of graphics processing units. It is primarily developed by Facebook's AI Research Lab. It is also one of the preferred deep learning research platforms built to provide maximum flexibility and speed. It is known for providing two of the most high-level features; namely, tensor computations with strong GPU acceleration support and building deep neural networks. Though still early in its inception, since its release in January 2016, PyTorch has quickly become a go-to library because of its ease in building extremely complex neural networks.

Some of the notable features include its simple interface, computational graphs, and dynamic library. It offers easy to use API which makes it very simple to operate and run like Python. This library, being Pythonic, smoothly integrates with the Python data science stack. This allows it to leverage all the services and functionalities offered by the Python environment. PyTorch provides an excellent platform which offers dynamic computational graphs, which you can change during runtime. This is beneficial when you are uncertain how much memory will be required for creating a neural network model as well as for debugging purposes and handling memory overflow.

One difference PyTorch possesses from its competitors such as TensorFlow is that it avoids static graphs, allowing developers and researchers to change how the network behaves instantaneously. This makes PyTorch more intuitive to learn when compared to TensorFlow. Since PyTorch is a native Python package by design. Its functionalities are built as Python. This is why all its code can seamlessly integrate with Python packages and modules. This Python-based library enables GPU-accelerated tensor computations and provides rich options of APIs for neural network applications.

Overall, PyTorch provides a simple solution to overcome all the challenges and has the necessary performance to get the job done. With its Python based infrastructure and intuitive, easy to use design, PyTorch continues to be favorable platform for developers and researchers in the field of Deep Learning and Artificial Intelligence.

Keras

Another leading high-level neural network API is Keras. It was created to be user friendly, modular, easy to extend, and to work with Python. The API was "designed for human beings, not machines," and "follows best practices for reducing cognitive load." Neural layers, cost functions, optimizers, initialization schemes, activation functions, and regularization schemes are all standalone modules that you can combine to create new models. New modules are simple to add, as new classes and functions. Models are defined in Python code, not separate model configuration files.

The biggest reason why Keras is widely popular and used is due to its ease of learning and ease of model building, as well as being user friendly. Keras offers the advantages of broad adoption, support for a wide range of production deployment options, integration with at least five back-end engines (TensorFlow, CNTK, Theano, MXNet, and PlaidML), and strong support for multiple GPUs and distributed training. Additionally, Keras is backed by Google, Microsoft, Amazon, Apple, Nvidia, Uber, and others.

The back end of Keras relies on a third party back-end engines to handle low-level operations, such as tensor products and convolutions. It comes default with a TensorFlow back-end, with the ability to change back-ends as Keras supports multiple back-ends. Keras also includes seven common Deep Learning sample datasets for developers looking to jumpstart their projects. These include cifar10 and cifar100 small color images, IMDB movie reviews, Reuters newswire topics, MNIST handwritten digits, MNIST fashion images, and Boston housing prices. This takes away the initial process of gathering large amounts of relevant data to begin training the model.

Drawbacks and Alternatives

Machine Learning and Deep Learning libraries and frameworks allow an easier and efficient approach to designing computational algorithms. These libraries often times require powerful CPU and GPU usage which may not be accessible for many. However, for quick application deployments, there exist online services that provide all the tools under one umbrella on a cloud-based platform. Amazon, an industry giant, offers cloud-based services for Machine Learning projects by its Amazon Web Services platform. Within this platform, there are a number of features pertaining to building ML applications. AWS Sagemaker offers notebooks for testing and verifying algorithms to create models. The training jobs can also be tuned through Hyperparameter Tuning Jobs that will run several tests to determine the optimal parameters to achieve your preferred results. The final models can then be deployed onto endpoints which can be directly accessed through applications to fetch real-time results. This ecosystem of features provides a seamless process for deploying ML based applications all from one service.

ii. Cloud Computing Platforms

Amazon EC2 *(Elastic Compute Cloud)*

EC2 provides a stellar solution to engineers who wish to obtain scalable computing capacity in the cloud. Not only are users able to customize the RAM that comes with their *instance*, but they are also able to pick between various CPUs and GPUs, which comes in handy while considering a Machine Learning focused project.

Regular use instances can have RAMs ranging from 1GB to even 64GB with GPUs included. Once these are initiated, users can use SSH to gain control of their "virtual computer on the cloud" and install the required packages and dependencies to run their achine learning models. Take, for example the p2.xlarge instance, which offers the following specs.

Specification	Value
Architecture	x86_64
Memory	64GB
Storage	Customizable

Network Performance	High
vCPUs	4

EC2s are known to be **highly** customizable and can be used virtually like a computer at home: for whatever purpose. This was both an advantage and disadvantage. Even though we required full freedom with our cloud computing solution, we did require a service from AWS that was completely dedicated to Machine Learning projects. EC2s may be powerful if required, but they often require the maintenance required for a regular computer.

We preferred a technology in which we could define our model and train it. We did not want to worry about installing dependencies, running shell scripts to keep our model training in the background, etc. Recording metrics such as training accuracy because of increasing epochs or validation accuracy with respect to time, etc. would have demanded the use of AWS CloudWatch, thereby increasing the labor an additional amount. Furthermore, we did not want to manually create our own HTTP based API to communicate with the model (which is imperative as we required prediction results in our React Native application).

AWS SageMaker offered a dedicated solution in which matched all our requirements as defined above. This was why we chose it over AWS EC2.

Amazon SageMaker

SageMaker is a dedicated, managed platform for machine learning. Built entirely for the purpose of catering to Machine Learning solutions, it offers a suite of services including (but far from limited to) *Managed Training Jobs, Notebooks*, *Managed Endpoints* (for interaction with GUIs), *Managed Hyperparameter tuning jobs.* As a team, we decided that, due to the array of managed features offered by SageMaker, it was the technology to use for this project.

We have included a holistic development journey using SageMaker, including the Machine Learning architectures used and relevant code.

Development Journey using Amazon SageMaker

Development on SageMaker was started using its Notebook Service, which initializes a Jupyter Notebook on the cloud and provides developers with a suite of SageMaker functionalities through a powerful Python SDK called boto3.

Multiple factors go into using this notebook to define, deploy and run an ML model. They are described below.

1. Initializing a Training Image

SageMaker excels at making Machine Learning accessible, and as such they provide various wrappers around common technologies, such as keras and tensorflow, through what is known as training_images. SageMaker has a library of such training images, untrained architectures ranging from simple Neural Networks, to Recurrent Networks, to Convolutional Neural Networks meant purposely for implementing Image Classification solutions. As such, for our project, the image-classification training image was used.

In the code snippet below, the notebook instance is given information on the *execution role* – a definition of what type of data the developer can use – and the *bucket*, which points to a location where our dataset is stored in Amazon's storage solution S3 (Amazon Simple Storage Service). Finally, the global variable training_image is set to image-classification.

In [1]:

```
%%time
import boto3
import re
from sagemaker import get_execution_role
from sagemaker.amazon.amazon_estimator import get_image_uri

role = get_execution_role()
print("Role = " + role)

bucket='senior-design-app-bucket' # customize to your bucket

training_image = get_image_uri(boto3.Session().region_name, 'image-classification')
```

```
Role = arn:aws:iam::467787479766:role/service-role/AmazonSageMaker-ExecutionRole-20200305T001560
CPU times: user 871 ms, sys: 145 ms, total: 1.02 s
Wall time: 8.08 s
```

2. Providing a Dataset

Our dataset of training images was stored on Amazon's cloud storage solution, S3. (This will be discussed in a later section.) The notebook, before any sort of training required information of the exact *uri* of where these datasets, train, validation, train_lst, and validation_lst are in the cloud.

In the snippet below it is made clear how these *uris*, prefixed with 's3' to indicate they are S3 resources are defined for use while defining and training the Image Classification Model.

The dataset locations train_lst and validation_lst, indicate the locations where the .lst files for the Train and Validation datasets are located. These list files include the ground truth for the images in their respective datasets, making it clear which class each image corresponds to.

In [2]:

```
# configuring the s3 uris for input data
s3train = 's3://{}/updated_dataset/train/'.format(bucket)
s3validation = 's3://{}/updated_dataset/validation/'.format(bucket)
s3train_lst = 's3://{}/updated_dataset/train_lst/'.format(bucket)
s3validation_lst = 's3://{}/updated_dataset/validation_lst/'.format(bucket)
```

3. Configuring Hyperparameters for Training Job

Amazon SageMaker does a tremendous job in giving developers access to sophisticated Convolutional Neural Networks through its Image Classification training image, on top of which it gives excellent control over the hyperparameters. In the code snippet below, we define the following hyperparameters.

- **num_layers** *(number of layers)*

- **image_shape** *(dimensions of each image in the dataset)*
- **num_training_samples** *(number of images in the training split)*
- **num_classes** *(number of defined classes in the dataset)*
- **mini_batch_size**
- **epochs**
- **learning_rate**

In [2]:

```
# The algorithm supports multiple network depth (number of layers). They are 18, 34, 50, 101, 152 and 200
# we'll use 50 layers
num_layers = "101"
# we need to specify the input image shape for the training data
image_shape = "3,244,244"
# we also need to zpecify the number of training samples in the training set
# for our dataset this is 8522 (skin dataset)
num_training_samples = "14707"
# specify the number of output classes
num_classes = "9"
# batch size for training
mini_batch_size =  "32"
# number of epochs
epochs = "100"
# learning rate
learning_rate = "0.01"
```

4. Deploying Training Job

After the dataset location is defined, and the hyperparameters of choice are also defined, it remains to start training the Image Classification CNN. This is done using the create_training_job feature of SageMaker. This is a managed service for training defined ML architectures on the cloud.

In the code snippet below, we have defined the training_params variable, which is essentially the training "definition" that is being discussed. It includes such sections as.

- **"AlgorithmSpecification"**
- **"ResourceConfig"** (defining the type of compute power is desired to train the model)
- **"HyperParameters"** (defining the hyper parameters)
- **"InputDataConfig"** (providing information on how to retrieve the training and validation datasets)

See the code snippet below to understand how these training parameters are set.

```
training_params = \
{
    # specify the training docker image
    "AlgorithmSpecification": {
        "TrainingImage": training_image,
        "TrainingInputMode": "File"
    },
    "RoleArn": role,
    "OutputDataConfig": {
        "S3OutputPath": 's3://{}/{}/output'.format(bucket, job_name_prefix)
    },
    "ResourceConfig": {
        "InstanceCount": 1,
        "InstanceType": "ml.p2.xlarge",
        "VolumeSizeInGB": 50
    },
    "TrainingJobName": job_name,
    "HyperParameters": {
        "image_shape": image_shape,
        "num_layers": str(num_layers),
        "num_training_samples": str(num_training_samples),
        "num_classes": str(num_classes),
        "mini_batch_size": str(mini_batch_size),
        "epochs": str(epochs),
        "learning_rate": str(learning_rate)
    },
    "StoppingCondition": {
        "MaxRuntimeInSeconds": 360000
    },
```

```
#Training data should be inside a subdirectory called "train"
#Validation data should be inside a subdirectory called "validation"
#The algorithm currently only supports fullyreplicated model (where data is copied onto each machine)
    "InputDataConfig": [
        {
            "ChannelName": "train",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3train,
                    "S3DataDistributionType": "FullyReplicated"
                }
            },
            "ContentType": "application/x-image",
            "CompressionType": "None"
        },
        {
            "ChannelName": "validation",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3validation,
                    "S3DataDistributionType": "FullyReplicated"
                }
            },
            "ContentType": "application/x-image",
            "CompressionType": "None"
        },
        {
            "ChannelName": "train_lst",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3train_lst,
                    "S3DataDistributionType": "FullyReplicated"
                }
            },
            "ContentType": "application/x-image",
            "CompressionType": "None"
        },
        {
            "ChannelName": "validation_lst",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3validation_lst,
                    "S3DataDistributionType": "FullyReplicated"
                }
            },
            "ContentType": "application/x-image",
            "CompressionType": "None"
        },
    ]
```

5. Hosting Endpoint on Cloud (real-time inference)

Once the model has been trained and its weights have been saved, SageMaker also offers a features to host the **trained** model on the cloud such that inferences can be obtained from it (from third party applications such as our React Native based smartphone application). This is a highly important feature, and helped us solidify our chance to use SageMaker, since we direly needed a service that could allow us to bridge our GUI with our Image Classification Model.

See the code snippet below to understand how first an endpoint_config is created, then used to establish an endpoint on the cloud. The endpoint_config is given information of the model that has been trained and is given details on what type of instance (or virtual computer on the cloud) is to host the inference-ready model.

In [6]:

```
endpoint_config_name = "skinnefy-training-version2-endpoint-config"
endpoint_config_response = sage.create_endpoint_config(
    EndpointConfigName = endpoint_config_name,
    ProductionVariants=[{
        'InstanceType':'ml.m4.xlarge',
        'InitialInstanceCount':1,
        'ModelName':model_name,
        'VariantName':'AllTraffic'}])

print('Endpoint configuration name: {}'.format(endpoint_config_name))
print('Endpoint configuration arn:  {}'.format(endpoint_config_response['EndpointConfigArn']))
```

```
Endpoint configuration name: skinnefy-training-version2-endpoint-config
Endpoint configuration arn:  arn:aws:sagemaker:us-east-2:467787479766:endpoint-config/skinnefy-training-version2-endpoint-co
nfig
```

iii. Datasets

To gather relevant images of all 9 skin diseases, we had to compile the data from multiple online sources. The data for the training and validation was acquired from both Kaggle and an online photo dermatology library called *Dermnet.*

Kaggle

The initial step before gathering all of the data required, is identifying and locating all the images the training model would need. Afterwards, the data would need to be gathered onto one location to process further. On Kaggle, the dataset called “DermMel” has 3 categories for determining whether a picture is of melanoma or not. The 3 directories are test, train_sep, and valid. Within each directory, there are 2 categories of pictures: Melanoma and NotMelanoma. Since the purpose of this application is to identify from 9 different skin diseases, the NotMelanoma images can be ignored. In total, 8,903 images of melanoma were gathered from this one dataset.

Another dataset on Kaggle called “Derma Diseases” also has 3 directories: test, train, and validation. Within each directory, there are another 3 subdirectories for the relevant images for: melanoma, nevus, and seborrheic_keratosis. For the purpose of this application, the nevus

classification was ignored because there were not sufficient amount images of nevus on Kaggle or any alternative data source. As a result, only melanoma and seborrheic keratosis were acquired from this dataset. In total, 439 images of melanoma and 413 images of seborrheic keratosis were obtained from this dataset.

It is important to note that these images will be stored in their own directories and will be labelled and processed accordingly prior to the training job.

Dermnet

Dermnet is the largest independent photo dermatology source dedicated to online medical education though articles, photos and video. Dermnet provides information on a wide variety of skin conditions through innovative media. On their website, is a collection of hundreds of dermatological disorders as well as hundreds of photos for their respective disorders. Before going into a specific disease, Dermnet has a master category which groups smaller sub-diseases together. For example, the class Acne and Rosacea photos contain sub classes for Acne – Cystic, Acne – Pustular, and many more. These sub classes in turn have hundreds of pictures that all relate to the main category. Unlike datasets on Kaggle, Dermnet images do not come in a prepackaged .zip file, therefore saving the entire catalog of images would take countless amount of time. To automate the process, we can utilize a python web-scraper that downloads and saves all images from a given website.

```
def genClassImages(class_url):
    """Fetch list of class images
    @arg class_url: web url
    @returns class_images: list of images
    """
    images = []
    urls = genClassCategories(class_url)
    print('- Found {} total sub-classes for class.'.format(len(urls)))

    for i, url in enumerate(urls):
        print('-- Fetching images from sub-class [{}/{}]'.format(i + 1, len(urls)))
        images.extend(genCategoryImages(url))

    return images
```

The above code snipped is used to extract all the classes of images from a single webpage. Within that class, all the images are found and compiled in a list which is then returned.

```
def genPageImages(url, image_list):
    """Finds all image links in a webpage and adds them to the image list.

    @arg url: web url; str
    @arg image_list: a list of image urls.
                     this will be modified in place.
    @return None
    """
    soup = soupify(url)
    thumbnails = soup.find_all("div","thumbnails")
    if thumbnails: ## there are thumbnails actually on the page
        for thumb in thumbnails:
            thumb_link = thumb.img['src']
            #use full image link instead of thumbnail link
            image_link = re.sub(r'Thumb',"",thumb_link)
            image_list.append(image_link)
```

This snippet of code expands on the image list created above and associated each image to its corresponding image link. Once these links are processed, they are then passed onto the main function which loops through the entirety of the dataset and saves each image on the predetermined directory on the local PC. The entire Web-scraper program can be found in the appendix of this report.

In order for Amazon SageMaker to be able to process and work with each image in the dataset when training for a model, each image is required to be the same size. Another way of describing this is that every image is required to be normalized. The term “normalization” actually encompasses more than just resizing, as the goal of this is to remove noise from every image such that each image falls into a range of intensity values that follows a normal distribution. However, a full discussion of normalization requires a deeper analysis in the subject of image processing, which is beyond the scope of this project. Thus, it is simply said that the reason for resizing every image in the dataset is so that when they are each processed by the SageMaker instance, the instance evaluates every image fairly; that is, with a weight that is similar to weights used for other images of the same class.

Because there are thousands of images in the dataset, it would be extremely impractical to resize every image individually. Thus, the resizing process is automated with the use of a custom-made Python script. This script makes use of the Python PIL module. PIL stands for “Python Imaging Library,” and is used to provide image processing capabilities to a Python script. New developments to the PIL module are available via Pillow, which is another Python image library that has been forked from PIL, where more active development is currently taking place. In the script used here, the PIL module is being used simply to resize every image.

The first step in the script is to define the file path where the original images are stored. This can vary depending on what the path is. The file path is stored as a string in the variable ‘path.’

Next, every image that is in this defined file path is stored into an array in the variable ‘dirs,’ with the help of the listdir() method provided by the Python provided os module. Afterwards, the custom resize() function is defined. The main part of this function takes place inside a for loop. This loop iterates through every item (image) in the ‘dirs’ array, and for each

item, it is first verified that the item is a real image by using the isfile() method. Once verified, the image is opened by the Image object provided by PIL. The resize() method from PIL is then used on the image. Here, the main parameter to pass to this method call is the desired size of the resized image. The desired size here is 244 x 244. The final step is to specify where the new resized image will be saved. A new file path is passed as a string in the PIL save() method call, along with the desired file format and quality.

This script as a whole is executed a certain number of times. The exact number of times it is used is the number of classes of images there are times 3 (one for the classes' training set, one for the validation set, and one for the test set). So for 3 classes, the script is run 3 * 3 = 9 times, and for 10 classes, it is run 10 * 3 = 30 times. Once all the resized images are obtained, they are pushed into the Amazon S3 bucket through the utilization of the AWS CLI.

```
from PIL import Image
import os, sys

path = "/Users/siddharthrajan/Users/siddharthrajan/dataset/validation/wart_viral_infections/"
dirs = os.listdir( path )

def resize():
    print("Resizing...")
    for item in dirs:
        if os.path.isfile(path+item):
            print("Processing: " + item)
            im = Image.open(path+item)
            imResize = im.resize((244,244), Image.ANTIALIAS)
            imResize.save("/Users/siddharthrajan/dataset/validation/wart_viral_infections/" + item, 'JPEG', quality=90)
    print("Done.")

resize()
```

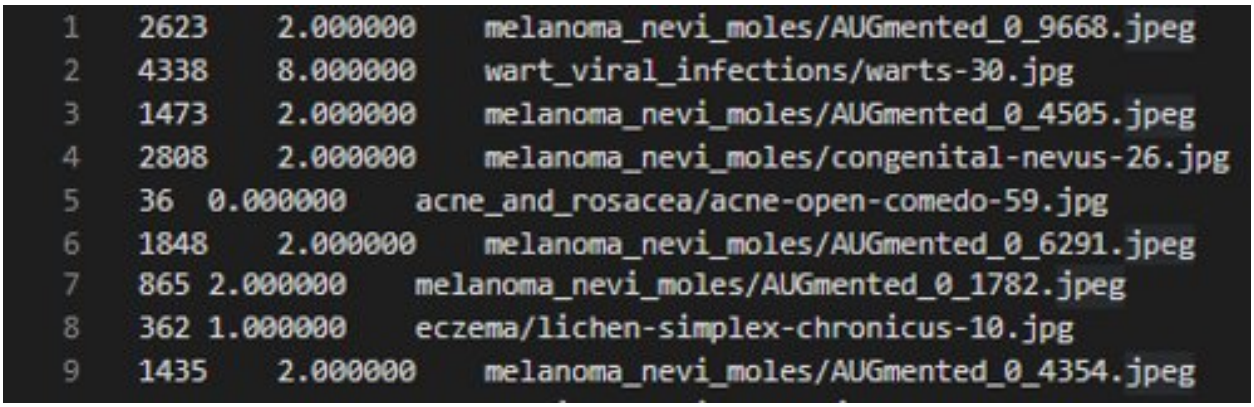
After the complete dataset is resized and finalized, the number of images in each class divided into the train, validation and test categories are shown in the table below. The 60-20-20 spilt is followed, where 60% of the data is the training set, and 20% is the validation and test sets.

Final Dataset

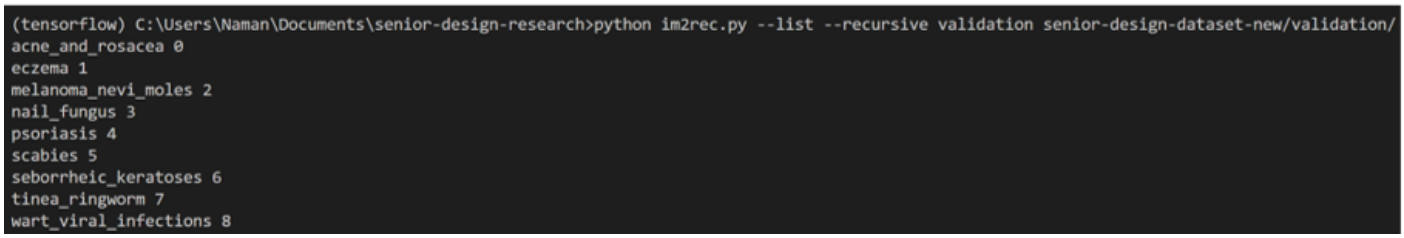
	Melanoma	Acne	Eczema	Seborrheic Keratosis	Warts	Scabies	Nail Fungus	Psoriasis	Tinea
Train	7182	527	929	1031	816	325	783	1056	977
Validation	2395	176	310	344	273	109	262	353	326
Test	2395	177	308	342	272	108	259	351	325
Total	11972	880	1547	1717	1361	542	1304	1760	1628

LST File

Yet another important step in processing the data is to create a .lst file. This file is required by our Machine Learning model to be implemented. A .lst file is a tab-separated file with three columns that contains a list of image files. The first column represents the image index. This has to be unique for all images since no two images can be at the same index. The second column is the class label index. In our case, this represents which skin condition the image falls into. Since we have 9 different classes, the class labels will range from [0,8]. Class 0, for example, could be acne, 1 could be eczema, etc…The third column is the relative image path. This is the location of the image on the PC.



The above image shows the first few contents of the lst tile. This file is thousands of lines long and includes the metadata for the entire dataset.



The lst file is generated by using an opensource program called im2rec.py provided by Apache. This script can be run directly from the command line with a given command and input parameters as shown above. Once the file is generated, the "ground truth" or the actual value of each of the class label index is established and displayed

b. UI Solutions

Developing an effective machine learning model is only one side of the coin that represents the fully functioning app. While this model is a major part of the backend, the other side of that coin consists of the frontend; the only part that the average user of this app will see and experience. In order for this app (and any app) to be practical and successful, it must have a functioning and visually appealing frontend. The frontend typically guides the user by showing

them the proper way of navigating through the app in an efficient and easy way. It is seen that while the backend allows the app to accomplish the main task it was made for, the frontend is what allows the app to actually be used for general purposes, and thus both are equally important. This section will discuss the thought process and all the procedures that were taken in developing the main app for detecting skin conditions.

i. Dedicated Website

Developing this particular app requires an extensive knowledge of web development. With web development, the most obvious choices that come into mind are HTML, CSS, and JavaScript. HTML (hypertext markup language) has been around for years and is essentially what all webpages are made from. Likewise, CSS provides the ability to bring different styles to any HTML page. While the importance of these languages isn't underestimated, there still needs to be a way to bring functionality to a webpage. That is where JavaScript comes in. JavaScript in recent years has become one of the primary languages associated with web development. Not only can it bring frontend functionality to a website, it can work in the backend as well. In recent times, numerous frameworks for JavaScript have been developed with the purpose of simplifying the web development process and limiting the amount of actual HTML and CSS written. A discussion of a few of them follow.

JavaScript-Based Web Frameworks

In today's time, JavaScript has a few popular frameworks that are used to make developing web pages much easier and more efficient. The purpose of a "framework" is to provide a library and tools intended to automate common tasks and functions. Some of these frameworks allow quick development of frontend components (such as textboxes and buttons) that would normally take many lines of HTML and CSS code to position and perfect. These same frameworks, and others, also allow quick calls to a backend server or function, typically for the purpose of retrieving some kind of information for a user. This would traditionally take more time with standard JavaScript, which commonly produces syntax errors and long periods of debugging. As a JavaScript-based framework, all these libraries and tools are written in JavaScript, and any development with these frameworks is done with JavaScript. When the development is finished, all the code is compiled into traditional HTML, CSS and JavaScript files, as these files and languages are the only things that the browser can interpret.

Common frontend frameworks include jQuery, React.js, Vue.js, Angular.js, Preact, Svelte, and Ember. Common frontend and backend frameworks include Express.js, Next, Meteor, Koa, Nuxt, and many more. For this project, only a few of these frameworks were viable candidates for the app, based on what development experience each group member had and which framework was unanimously agreed upon as the "easiest." What follows is a discussion of some of these choices.

Angular.js

Angular.js is a frontend JavaScript framework that has been popular in recent years, although its favorability has become relatively lower due to the increasing demand for React in the industry.

Angular provides the ability to extend HTML so that pages can be used and manipulated in a dynamic fashion. Specifically, this means that Angular provides the ability to create custom tags in HTML. This is useful when it is desired to create custom reusable components (tags). In addition, Angular allows the separation of the application's business logic from the "view" (HTML pages). This is convenient because when a file is mixed with both the frontend HTML and backend JavaScript, it can become tedious and hard to follow.

An important property of Angular is that it is a declarative framework, as opposed to an imperative framework. Being declarative means that when implementing features in the application, all the required behaviors are specified and executed within the HTML, with occasional calls to backend JavaScript functions. On the other hand, being imperative means that the HTML only consists of a few of the standard tags such as which don't specify exactly what their purpose is. More emphasis is given to the backend JavaScript which would be used to initialize these tags.

From the preceding description, it is seen that Angular is a framework that is very centered around HTML. It is most appealing to those who are already very experienced with HTML and would like to enhance its capabilities to suit their needs. Of course, all the functionality of CSS is still available when using Angular.

As for the final skin-condition detecting app, the decision was reached in favor of **NOT** using Angular. There were numerous reasons behind this, the main one being a lack of experience with the framework. Other than that, an important factor in this decision was that there was a mutual agreement among the group towards not basing all of the application's business logic around a series of heavy HTML pages. Although this framework could be relatively quick to pick up, there was the possibility of having to spend long periods of time debugging the logic should errors have occurred. An alternative framework was desired; particularly one that at least some of the group already had experience in.

React.js

React.js is another frontend JavaScript framework. It has grown to be one of the most popular ways of developing functional user interfaces.

React is significantly different from Angular. As mentioned before, Angular places its emphasis on the HTML, while React does just about the opposite. Almost no additional HTML is added to the HTML files, aside from the standard <html><head>…</html> tags. Instead, only one pair of tags is typically added to the HTML document. This tag is given an id or class name such as "root." From here on, all the application's business logic, both frontend and backend, take place in various JavaScript files.

These JavaScript files consist of either one of two entities: a function or a class. These functions and classes are known as React Components, and they are the main building blocks of React applications. The body of these functions and classes contain one section that "returns" some code segment (it is placed inside a return block). What exactly is returned depends on the

purpose of the React Component. If the component is a special kind of text box for example, the implementation of a textbox is placed in the return block. This leads to the question of how exactly that text box is implemented. In React, a form of syntax that is similar in appearance to HTML is used. Despite being similar, this syntax couldn't be more different. This syntax is known as JSX, and it is an extension to JavaScript. The purpose of writing code in JSX is to make it easier to visualize how something is being implemented. When JSX code is compiled afterwards, it is converted into standard HTML.

Having reusable function or class components is only one benefit of using React. The second (and very important) benefit are the concepts of state and rendering. React provides the powerful ability to have each class component maintain a particular state, which is a JavaScript object that details specific properties of the page at a certain moment in time. Common states include the number of times a button was clicked or what word was typed in a text box. When a state needs to be updated, the convenient setState() method is called. The setState() method brings forth the concept of rendering in React. React provides the convenient feature of rendering a page once and only rendering it again when a certain state changes. When it renders again, only the changes from the updated state are reflected on the page.

From this discussion it is clear that there are many benefits of using React. This is therefore the most ideal choice for developing the application. It should be noted, however, that React is mainly suited for developing webpages/websites, which is certainly not the same as developing a mobile app. The developers of React actually created a solution for this; the result is an extension of React called React Native, which is specifically made for using the functionality of React to develop native apps for iOS and Android. React Native is the framework of choice that was finally agreed upon for this project, and a more in-depth discussion of React Native follows in the next section.

ii. Mobile Applications

Front-End Solutions

React Native

We used the popular JavaScript based framework React Native to develop our application. Built upon the already existing web framework React (which also used JavaScript to develop websites) React Native renders code in your smartphone. Developers that are accustomed to React can essentially code and develop using the same skillset but on different hardware.

Using React Native, developers can produce apps compatible with both iOS and Android devices and boasts various advantages in the form of its expansive APIs and the extent to which the framework can give developers control over smartphones in their applications. One of the advantages the technology has to offer is that since the framework is based on JavaScript, developers need not rebuild their application to see any reflected changes. Just by refreshing their application, or in newer iterations just by saving their source file, changes in the application will appear instantaneously. Another major advantage of the technology is in its code re-usability and ease of debugging. It becomes obvious when considering codebases that React Native offers the advantages of not having separate sources for iOS and Android. Rather, it reduces the development

effort into just one codebase that can be debugged by anyone who is a React Native expert. And these changes will be reflected in both the iOS and Android versions of the application.

1. Setup and Usage

React Native projects can be initialized using the ***Expo*** managed workflow. Expo is essentially a toolchain surrounding React Native and offers additional APIs alongside various other developmental features. Features such as cross-collaboration between different devices, development on multiple devices at the same time, and cloud based code sharing are all made possible using Expo.

As Expo is an npm module, it can be installed on the terminal using the command

```
npm install expo-cli --global
```

Following this, any managed react native application can be ran using the command

```
expo start
```

At this stage, the codebase can be edited, and any changes made are updated live on an emulator or (if you prefer to develop directly on a device of your choice) your device. This is made very convenient and easy by the Expo QR code system, wherein any compatible smartphone can scan a QR code, and live development code and be rendered on it! (and of course, it will update as the codebase you are writing updates!)

Raw, out of the box React Native **can** definitely be used to make a full-fledged application, but third party modules can enhance this experience by providing reliable, pre-written functionality to the application you are developing. Not only does this prevent you from re-inventing the wheel, but it helps optimize your code as these modules are tested thoroughly before they are able to be downloaded publicly using npm, the node package manager. Below is a table of important modules that were used to develop our application.

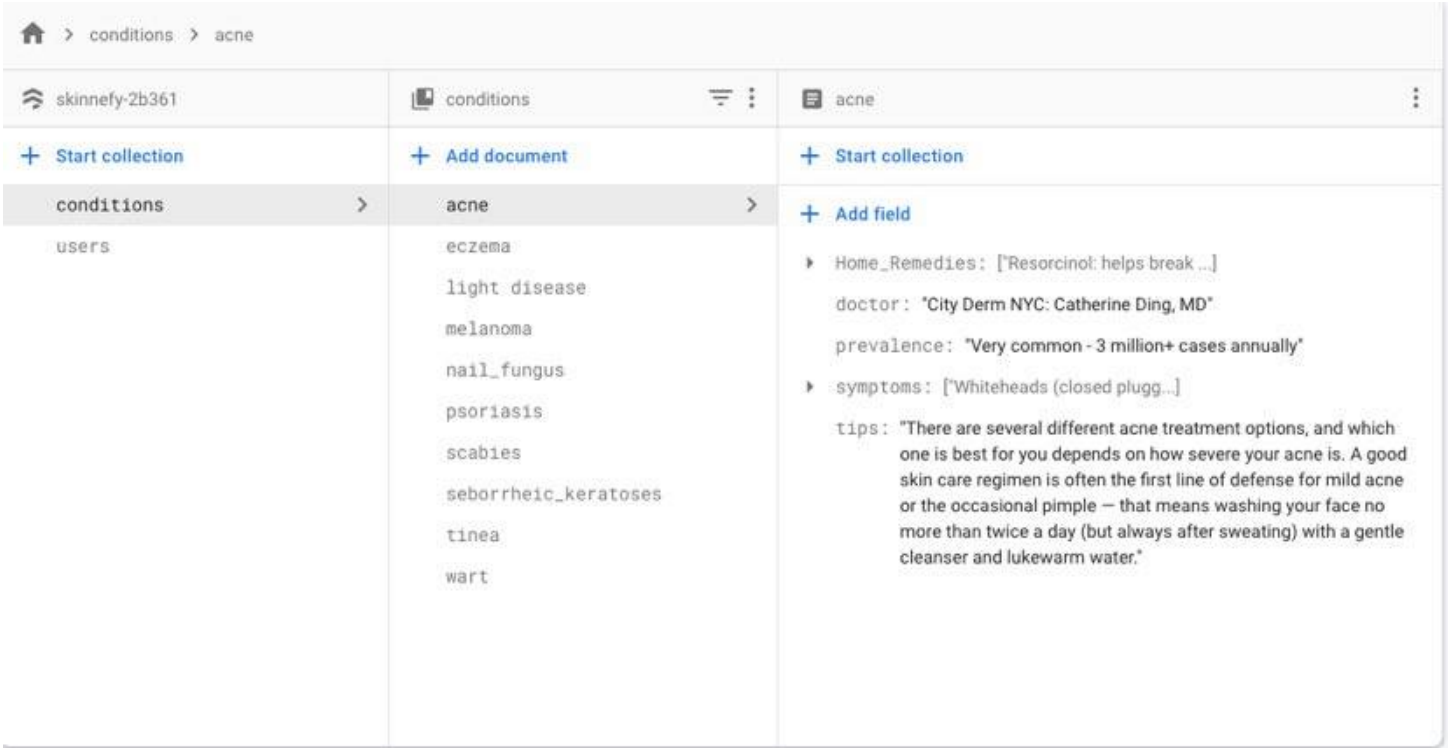
Name of Module	Description
react-navigation	The core navigation module that allows users to **navigate** through the application. This provides our application with tabs to press and switch to different screens, properly functioning back buttons to route to previous screens, and so much more.
react-native-elements	Not all UI components that developers make are appealing and aesthetic. Because of this, react-native-elements offers pre-developed, appealing UI components that are extremely adaptable and reusable. These include Avatars, Lists, Tables, Badges, Buttons, Text Fields, etc.
firebase	Gives our application access to Firebase, a google back-end service featuring managed services for database, authentication, and storage. ***Will be discussed in further sections.***
aws-sdk	A JavaScript based AWS SDK allowing us to tap into the SageMaker endpoint (discussed previously) hosting our trained Convolutional Neural Network. This is the module that makes the communication between our application and our trained model possible. ***Will be discussed in further sections.***

Back-End Solutions

Firebase

The back-end of the application is supported by Firebase. Firebase is a Backend-as-a-Service — BaaS — that is part of the Google Cloud Platform. It frees developers to focus crafting fantastic user experiences, without having to write APIs, set up external server, or create a SQL database from scratch. Firebase offers a real-time NoSQL database. Most databases require making HTTP calls to get and sync the data. When applications connect to Firebase, they are not connecting through HTTP, rather through a WebSocket, which are much faster than HTTP. This means individual WebSocket calls are not needed because one connection is sufficient to retrieve data. Data is synced through that single WebSocket as fast as the client's network can carry it.

Firebase sends new data as soon as there are updates. When a client saves a change to the data, all connected clients receive the updated data almost instantly. For this application, the different skin conditions, along with their relevant information will be populated within the database. The image below shows a breakdown of how information is organized in the database.



All the conditions have relevant information stored within the "document" as a "field". To further explain how a NoSQL database works, we can consider the example of modeling the schema for a simple skin condition and information database. In a NoSQL database, the condition is usually stored as a JSON document. For each condition, the Home Remedies, Doctor, Prevalence, Symptoms, and Tips are stored as attributes in a single document. In this model, data is optimized for intuitive development and horizontal scalability.
To further organize the data, attributes that have multiple elements can be stored as arrays. For example, the Home_Remedies, and Symptoms fields are stored as:

Home_Remedies: {Home_Remedies[0]…Home_Remedies[n]}
Symptoms: {Symptoms [0]…Symptoms [n]}

This helps with organization and formatting when displaying the text on the front-end.

There are two use cases for accessing the database. The first revolves around pulling the document data just once. This means that the information is only queried once. From this method, a listener or a document subscriber is not established. If any further changes are made to a document, they do not update real time on the application. For example, the remedies tab on the UI has preestablished information on it that does not get updated. So for that specific case, the information would only need to be fetched once and displayed onto the tab. The relevant code for this method is as follows:

```
export const useDiagnosisOnce = (postDocName) => {
    const [docState, setDocState] = React.useState({ isLoading: true, data: null });
    const compileData = async () => {
        var diagnosisDoc = await db.collection("diagnoses").doc(postDocName).get();
        setDocState({
            isLoading: false,
            data: diagnosisDoc.data(),
        });
    }
    React.useEffect(() => {
        compileData();
    }, []);
    return docState;
}
```

We can also subscribe to a document in a collection. “Subscribing” refers to attaching a listener to a document and expecting real-time updates if document state is changed. Whenever there is an update to the database, the changes are relayed back to the application and displayed in real-time. This could especially be useful for displaying profiles since the number of diagnoses taken are constantly changing as well as new images being saved to the user’s profile. This can be implemented with the following snippet of code:

```
export const useFirestoreDoc = (collectionName, docName) => {
    const [docState, setDocState] = React.useState({ isLoading: true, data: null });
    React.useEffect(() => {
        return db.collection(collectionName)
            .doc(docName)
            .onSnapshot(doc => {
            setDocState({ isLoading: false, data: doc.data() });
        });
    }, []);
    return docState;
}
```

Aside from the database, Firebase Authentication provides easy-to-use SDKs, and ready-made UI libraries to authenticate users to applications. Firebase authentication has a built-in email/password authentication system and also supports OAuth2 for Google, Facebook, Twitter and GitHub. For this application, the email/password authentication method was implemented. Email/password authentication has functions to register new users, sign in existing users and sign out a signed-in user.

The sign in process works by taking the email/password credentials from the user and passing them onto the Firebase Authentication SDK. It will then verify those credentials and return a response to the user. After a successful sign in, we can access the user's basic profile information and control the user's access to data stored in other Firebase products.

```
export const useAuth = () => {
    const [state, setState] = React.useState(() => {
        const user = firebase.auth().currentUser;
        return { initializing: !user, user, }
    })
    function onChange(user) {
        setState({ initializing: false, user });
    }
    React.useEffect(() => {
        // listen for auth changes
        const unsubscribe = firebase.auth().onAuthStateChanged(onChange);
        // unsubscribe to the listener when unmounting
        return () => unsubscribe();
    }, []);
    return state;
}
```

The above portion of code shows the steps to setting up authentication listener. The application continually checks the user status. If the status changes from signed in to signed out, then the application is closed and hidden. If it changed from signed out to signed in, this indicated a log in, and therefore, the application is displayed.

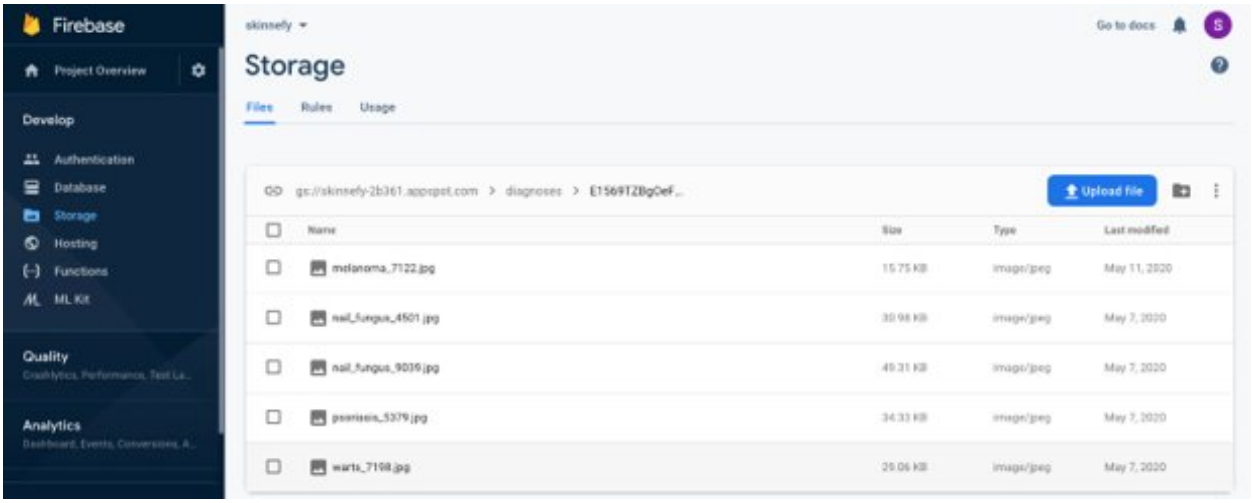
```
export function logout() {
    auth.signOut().then(() => console.log("User Signed out"))
    .catch(() => console.log(error.message));
}
```

The code above shows the functionality for logging out. The function is only called when the logout button is pressed in the application.

```
export function loginWithPassword(email, password) {
    return auth.signInWithEmailAndPassword(email, password);
}
```

There is also a similar function that is called when the “sign-in” button is pressed on the application. This is the procedure for logging in.

Additionally, Firebase offers cloud storage which allows storing files in a Google Cloud Storage bucket and is made easily accessible through Cloud Firestore. Files such as images, audio, video, or other user-generated content can be easily stored and accessed. For the purpose of this application, only images previously taken by the user are stored and displayed on the user profile. This feature is especially useful if a user wants to consult with a doctor and show them the history or progress of their skin and the pictures they have taken on this application. The contents of the User Profile tab on the UI depend heavily on this feature since the user profile should have access to the user’s history.



AWS SDK (Software Development Kit)

The Amazon Web Services SDK offers all the services the cloud platform has to offer but bundled into one installable package depending on what platform a certain develop is coding on. For our case, we chose the JavaScript SDK as our project pertained to React Native. This was made available to us through, once again, npm, (node package manager), by running the command

```
npm install aws-sdk
```

Following the installation of this module in our application development environment, we are given the ability to access SageMaker endpoints using the invokeEndpoint method as described below.

```
var params = {
  Body: Buffer.from('...') || 'STRING_VALUE' /* Strings will be Base-64 encoded on your behalf */, /* required */
  EndpointName: 'STRING_VALUE', /* required */
  Accept: 'STRING_VALUE',
  ContentType: 'STRING_VALUE',
  CustomAttributes: 'STRING_VALUE',
  TargetModel: 'STRING_VALUE'
};
sagemakerruntime.invokeEndpoint(params, function(err, data) {
  if (err) console.log(err, err.stack); // an error occurred
  else     console.log(data);           // successful response
});
```

As seen in the example code snippet above, the invokeEndpoint method requires a params JSON that describes the *body* (in our case ByteStream of the image we are sending to our trained model for inference), *endpoint name*, and *content type*, among other relevant fields, once these are defined, its evident how easy the AWS SDK makes invoking an endpoint: one function call.

This example code, of course has been adapted to our use case. In the code snippet(s) below it's clear how we are parsing the incoming images into Blob files (Binary Large Objects), and using the invokeEndpoint method to run inference on the images.

Lines 37 and 38 demonstrate how images taken by the users are converted to Blobs, which are then – as mentioned – passed onto the invokeEndpoint method in line 39.

```
32  export function useEndpoint(image) {
33      const [queryState, setQueryState] = React.useState({ isLoading: true, data: null, err: null });
34
35      const sendAndRetrieve = async () => {
36          try {
37              const response = await fetch(image.uri);
38              const blob = await response.blob();
39              sagemakerruntime.invokeEndpoint({
40                  Body: blob,
41                  EndpointName: endpoint_name,
42                  ContentType: "application/x-image",
43              }, function(err, data) {
44                  if(err) {
45                      setQueryState({
46                          isLoading: false,
47                          data: null,
48                          err: err.message
49                      })
50                  }
51                  else {
52                      let prediction = prepareResponse(data.Body);
53                      setQueryState({
54                          isLoading: false,
55                          data: prediction,
56                          err: null
57                      })
58                  }
59              });
60          }
61          catch (error) {
62              console.log(error)
63          }
64  }
65      React.useEffect(() => {
66          sendAndRetrieve();
67      }, []);
```

RESULTS

a. Machine Learning Model

The following table shows the parameters and results of all training and tuning jobs initiated with Amazon SageMaker.

Tuning Job*	Training Job Name	Epochs	# Classes	Learning Rate	Mini Batch Size	# Layers	Training Accuracy	Validation Accuracy
N/A	skinclassification-test1-2020-03-18-02-02-53	30	3	0.01	64	50	0.991	0.900
skin-condition-tuning-job-4	skin-condition-tuning-job-4-022-2ec93c08	30	3	0.02	64	50	0.996	0.900
skinnefy-model-tuning-job-v5	skinnefy-model-tuning-job-v5-010-bf49c55a	30	10	0.001	32	50	0.957	0.665
N/A	skinnefy-training-version1	75	10	0.01	32	110	0.795	0.643
N/A	skinnefy-training-version2	75	10	0.01	32	110	0.983	0.639
N/A	skinnefy-training-job-version-3	100	9	0.01	16	101	0.986	0.669
N/A	skinnefy-training-job-version-4	60	9	0.01	8	101	0.974	0.753

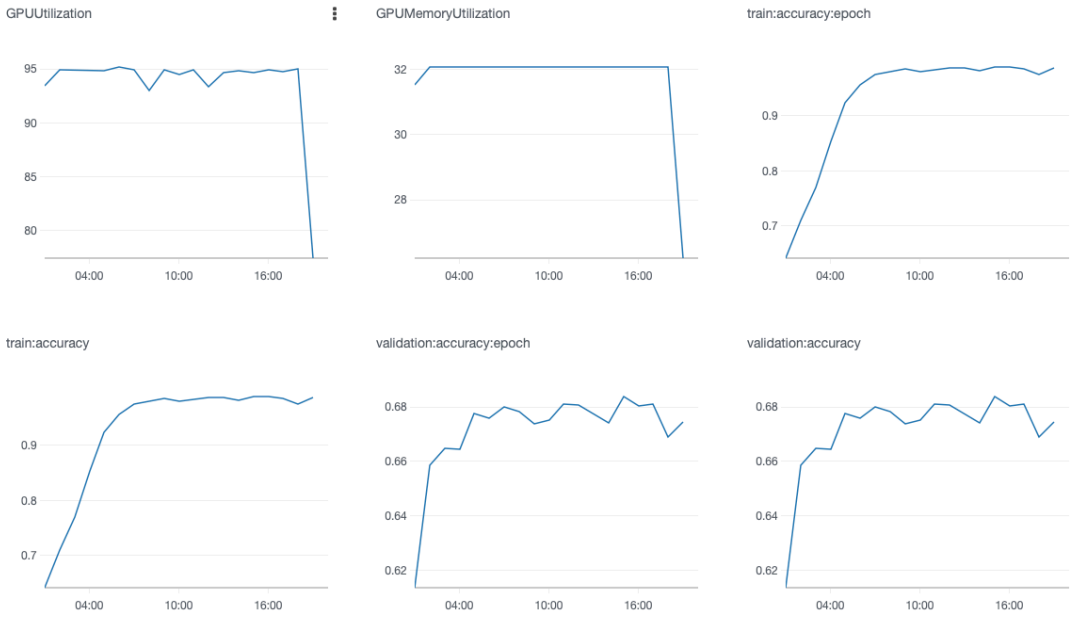
* Only the best training job from every tuning job is shown.

The training jobs began with the training of only three conditions; namely melanoma, acne, and warts. The purpose of this was to get used to the environment and techniques of Amazon SageMaker. It was noticed that with three classes of conditions, the results were very accurate: there was more than 95% accuracy with the training set and about 90% accuracy with the validation set.

Afterwards, the number of classes was increased to 10, as it was intended that the final app should be able to distinguish between 10 different conditions. However, the end results for the first training job with 10 classes was not what was expected. For the most part, the training accuracy was still very high being more than 95%, but unfortunately the validation accuracy went lower than expected, being around 65-70%. Techniques such as lowering the batch size and increasing the number of layers and epochs were attempted for subsequent training jobs with 10 classes, but the results didn't improve much.

At this stage, one of the 10 classes/conditions, namely light disease, was removed. This was because the group noticed that the images belonging to this class were simply not reliable. Lots of the images in this class were taken from different angles along with bad lighting. It was a group decision to remove this class and see if the results improved. However, it was seen that this also didn't do much to improve the training and validation accuracy, as the training accuracy was still high and the validation accuracy was still between 65-75%.

The following figure is a visual depiction of the training job “skinnefy-training-job-version-3” shown in the table earlier. The first two graphs are metrics relating to the SageMaker instance that performed the training job. The following four graphs are of more interest, as they show the training and validation accuracy. When the training job first starts, both accuracies are low, but over time as the training progresses, both accuracies converge around a particular value. In general, the training accuracy converged to a higher value than the validation accuracy.



b. UI

i. Remedy Tab

The leftmost tab of the app consists of a section that displays information about common skin conditions, specifically the ones that the SageMaker generated model was trained on. Information for about 10 different conditions is available for analyzing. The design style that was chosen for this section was a card layout, where different card windows, one for each skin condition, are visible one after another. The user can scroll down to see all the cards. Each card consists of an image of the specified condition, as well as a brief description of it. Upon pressing one of the cards, the user is taken to a second page where additional details about the condition can be viewed. These details include the prevalence of the disease, a doctor that is available for further consultation, common symptoms and home remedies, and general tips for treatment.

The way that the card layout was implemented is a multistep process. An array of objects is first defined in React where each object has key/value pairs that specify the full condition name, a URL to its image, the condition description, and a database reference string. A React class called “RemediesScreen” is then defined. Because this is a class, it must contain a render() method, which will return the JSX that forms the screen. The class also has a constructor which calls super(props). In the render method’s return block, a <ScrollView /> tag is first defined along with props passed to it which define its style. The purpose of this is to allow the user to be

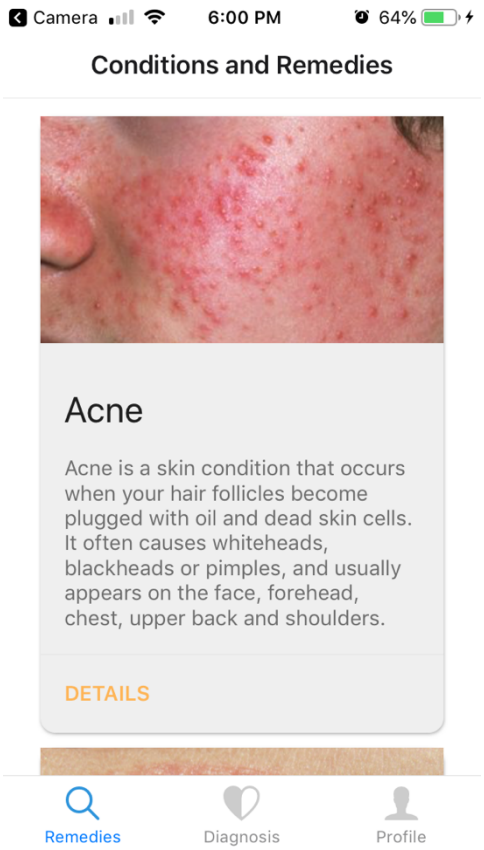
able to scroll through this page when the Remedy tab is selected. Within the opening and closing <ScrollView> tags, the .map() method is called on the array that was defined earlier. The .map() method allows specific information from each array element to be used in whichever fashion desired. In this case, it is used to display the desired condition information in a card. Therefore, the .map() method's return block returns a <Card /> element. Since .map() is an iteration method, one <Card> is generated for each condition in the array. Within the opening and closing <Card> tags, various other card-related tags provided by the 'react-native-cards' module are used to make each card display the appropriate information. A visual of this code is shown in the following figure.

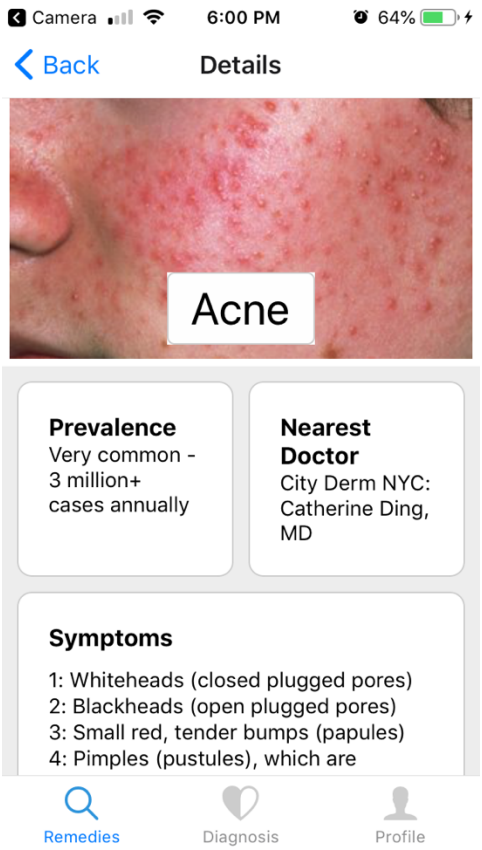
```
render() {
    return (
        <React.Fragment>
        <ScrollView style={{ flex: 1 }} contentContainerStyle={styles.container}>
            {prepared_conditions.map((condition, key) => {
                return (
                    <Card style={{
                        backgroundColor: "#efefef",
                        borderRadius: 10, elevation: 2,
                    }}
                    key={key}>
                        <CardImage
                            source={{uri : condition.image}}
                            style={{
                                height: 150,
                            }}
                        />
                        <CardTitle title={condition.name} />
                        <CardContent text={condition.description} />
                        <CardAction separator={true} inColumn={false}>
                            <CardButton
                                onPress={() => {this.props.navigation.navigate('Details', {
                                    query: condition.ref,
                                    displayTop: condition.name,
                                    description: condition.description,
                                    image: condition.image,
                                })}}
                                title="Details"
                                color="#FEB557"
                            />
                        </CardAction>
                    </Card>
                )
            })
            }
        </ScrollView>
```

The next part of the Remedy tab involves implementing the details page, where the additional information about each condition is shown. This page involves more steps, such as taking the condition associated with the card that was pressed and querying the database with it, as well as making use of React states and React router. First, a React class called "DetailsScreen" is defined. Like before, a constructor and render() method is provided. This class also contains another method called componentDidMount(). This method is provided by React and is meant to implement any steps that must take place when the page first loads. In the constructor, after calling super(props), the class's state is set to make isLoading true and data null (undefined). When isLoading is true, the page will only show a loading animation. This goes away only when data from the database for the desired condition is retrieved. The database call to retrieve said data takes place in componentDidMount(). In order to make the call, the ability of React router must be understood and taken advantage of.

```
componentDidMount() {
    db.collection('conditions').doc(this.props.route.params.query).get().then(doc => {
        this.setState({
            isLoading: false,
            data: doc.data(),
        });
    })
}
```

React router provides a way to navigate between and transmit information between different pages in any React project. It is used here to transmit the database reference name for the condition from the “RemediesScreen” class to the “DetailsScreen” class. In the “RemediesScreen” class, the button on each card has a “onPress” prop which accomplishes this. Pressing the card button both navigates to and transmits the appropriate information by calling “this.props.navigation.navigate().” The navigation prop is provided by default to a React class. A string representing the destination page and the information to send are given as parameters to this method. The destination string that is used here must match what is specified in an external file known as a router file. The router file makes use of react stacks to specify the structure of navigation between different pages in the app. Once the “DetailsScreen” class has received the information, it can be accessed from “this.props.route.params.” After this, the rest of the details page is made by just formatting and styling the retrieved information so that the page is easily understandable and visually appealing. The end result of the remedy tab is shown below.





ii. Diagnoses Tab

The primary tab of the user interface allows users to interact with the application using their cameras and receive a diagnosis for their skin condition. In laymen terms, users can use this tab to take a picture and receive a skin condition which, of course, is a result of the Image Classification Model hosted on AWS SageMaker. The Diagnoses Tab is comprised of **two** screens, one which hosts the code for the displaying the camera, and the other which hosts the code for displaying the results in a comprehensive bar-chart format. These two, respectively, are referred to as the *Camera Screen*, and the *Results Screen*.

The design behind the *Camera Screen* is minimalistic. The view provided by either the back or front camera (depending on whichever is chosen by the user) fills the entire screen, while a helper text sits atop the *Take Picture* and *Flip* Camera icons. The application is given access to the Camera after it first prompts the user to allow hardware permissions. Following this, the Camera API is enabled and can fetch real time images from the smartphone camera. Below, in the short code snippet for the *Camera Screen* it can be seen how the term <Camera> encapsulates everything. This is essence, indicates that whatever GUI is present in the *Camera Screen* (such as the helper text, and the two functional icons), depend **entirely** on the functionality of the camera. In addition, it indicates that functionality of the Camera is essential for the application to run. Please see the Code Snippet below.

```
<Camera style={{ flex: 1, }} type={type} ratio={CAMERA_RATIO}
  autoFocus={false} pictureSize={size}
  ref={ref => { camera = ref; }} autoFocus
  onCameraReady={getPictureSizes}
>
  <View style={{flex: 1, flexDirection: "column", justifyContent: "flex-end"}}>
    <Text style={{fontWeight: "bold", padding: 25, fontSize: 15, color: "#fff", alignSelf: "center",}}>
      Take a picture of your skin condition
    </Text>
    <View style={{paddingBottom: 20, flexDirection: "row", justifyContent: "space-evenly",
        alignItems: "center",}}>
        <TouchableOpacity
            onPress={() => {
            camera.takePictureAsync({
              base64: true,
              compression: 0.0,
              onPictureSaved: onPictureSaved});
            }}
        >
            <Ionicons
              name='ios-radio-button-off'
              color="#fefefe"
              size={70}
            />
        </TouchableOpacity>
        <TouchableOpacity
            onPress={() => {
            setType(
            type === Camera.Constants.Type.back
                ? Camera.Constants.Type.front
                : Camera.Constants.Type.back
            );
        }}>
            <Ionicons
            name='ios-refresh'
            color="#fefefe"
            size={35}
            />
        </TouchableOpacity>
    </View>
  </View>
</Camera>
```

In the code snippet above, it is also evident how users can “press” on the buttons for flipping the camera and taking an image. In lines 67 and 81, two separates <TouchableOpacity> tags are declared, which logically enough, act as opacities that can be “touched”. Under these tags are wrapped icons that, when pressed are commanded to do the functionality declared under the onPress method of each of their parent <TouchableOpacity> tags. Another point of interest is in line 72, which defines the functionality behind taking a picture, which is blatant by the name of the function onPictureSaved. Details of this function are given in the code snippet below.

```
42    const onPictureSaved = photo => {
43      navigation.navigate('Results', { data: photo })
44    }
```

This simple piece of code is very meaningful, as it makes use of the navigation function, which is fundamental towards the switching of screens in the application. In line 43, it is made clear that when the picture is taken and saved (in temporary application cache) by the camera, the navigation function is called to switch the screen to the *Results Screen*, which is given details of the recent image taken (the parameter photo seen in line 42) through the data field in line 43. This variable photo will then be used in the *Results Screen* to display the picture that was just taken, and most importantly to query the Image Classification Model hosted on SageMaker, using the AWS JavaScript SDK.

The *Results Screen*, as previously mentioned, provides its predictions on the users’ skin condition. The structure of this screen is as such: a picture of your skin condition (taken from the camera) is displayed on top, while its prediction is displayed in the form of a bar chart in the bottom half. The prediction that is received from the Machine Learning model is displayed neatly on a bar chart showing the top two predictions. This user interface feature is implemented using the <BarChart> tag shown in the code snippet below.

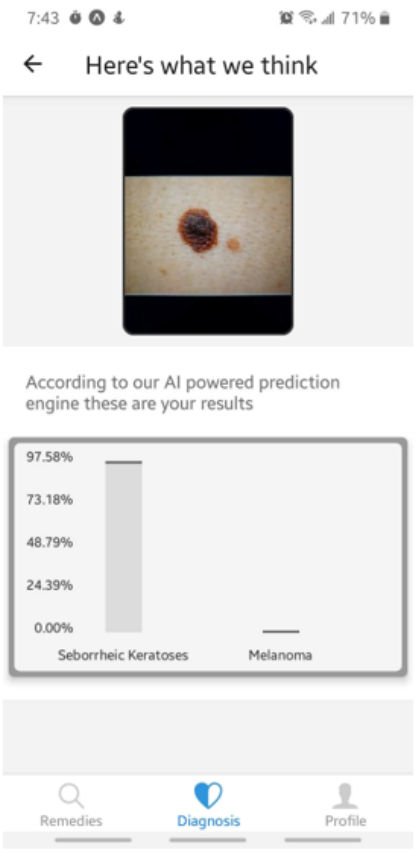
```
42    <BarChart
43        // style={graphStyle}
44        fromZero={true}
45        segments={4}
46        withInnerLines={false}
47        data={{
48            labels: data.labels.slice(0, 2),
49            datasets: [
50                {
51                    data: data.dataset.slice(0, 2).map(val => {
52                        return val * 100;
53                    }),
54                },
55            ],
56        }}
57        yAxisSuffix="%"
58        width={Dimensions.get('window').width - 20}
59        height={200}
60        chartConfig={{
61            backgroundColor: '#1cc910',
62            backgroundGradientFrom: '#fff',
63            backgroundGradientTo: '#999',
64            decimalPlaces: 2, // optional, defaults to 2dp
65            color: (opacity = 255) => `rgba(0, 0, 0, ${opacity})`,
66            style: {
67            borderRadius: 16
68            },
69
70        }}
71        style={{
72            borderRadius: 5,
73            padding: 5,
74            backgroundColor: "#999",
75            elevation: 5,
76        }}
77    />
```

Line 51 describes what data is displayed on the bar chart. Using the .slice method, the prediction array is sliced such that only the top 2 results remain. This is then displayed. This however, begs the question: how is the prediction array being retrieved. This is also done in the *Results Screen* through a function called useEndpoint, which queries the Image Classification Model using the picture just taken from the camera.

```
12      const { isLoading, data, err } = useEndpoint(route.params.data);
```

Again, useEndpoint is the function that queries the AWS SageMaker *Endpoint* API and retrieves prediction values from models hosted on SageMaker. This has been discussed extensively in the *AWS SDK* section, under *Back-end Solutions*, in *Mobile Applications.*

Below are some screenshots of the graphical components that are showcased in the Diagnoses Tab.



iii. Profile Tab

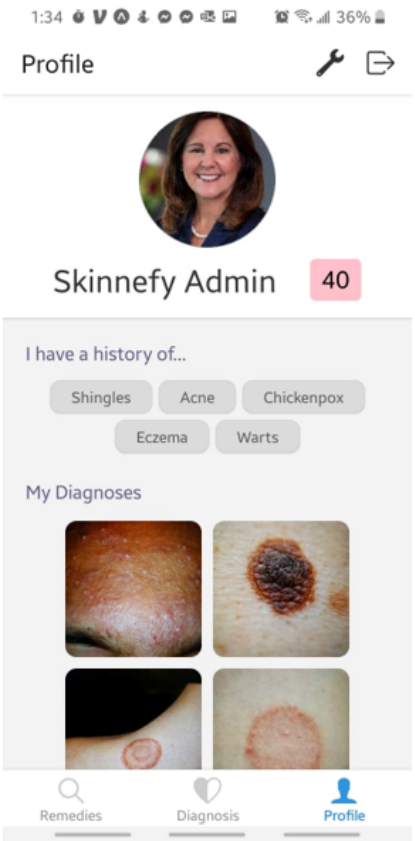
The profile tab is the rightmost tab and serves the purpose mainly of displaying the user's information. The UI of this screen is minimalistic; it displays the user's avatar at the top within a circular border and indicates their age and gender. Next, the tab also includes various Badges that indicate the user's previous pertinent medical history. Finally, a gallery of their previous skin conditions (all the images they have taken using the Diagnoses Tab) are displayed as a form of record keeping.

Of course, all the data that is displayed here is pulled from our firebase database relating specifically to user records. See the code snippet below that explains how the array of user conditions are mapped into user interface elements also known as <Badges>.

```
100     <Text style={{fontSize:15, color: "#5a5d81"}}>
101         I have a history of...
102     </Text>
103     <View style={{
104         paddingVertical: 10,
105         flexDirection: "row", flexWrap: "wrap",
106         justifyContent: "center"
107     }}>
108         {
109             data.history.map((x, i) => {
110                 return (
111                     <Badge
112                         badgeStyle={{
113                         backgroundColor: "#dbdbdb",
114                         borderColor: "#cccccc",
115                         borderWidth: 0.5,
116                         elevation: 1,
117                         padding: 14,
118                         margin: 3,
119                         }}
120                         key={i}
121                         value={x}
122                         textStyle={{
123                             color: "#555"
124                         }}
125                     />
126                 )
127             })
128         }
129     </View>
```

Line 109 describes how each skin condition is displayed as a <Badge> component, using the .map method that is provided for all arrays/lists in JavaScript. .map essentially acts as an iterator and provides access to each value in the list, along with its index.

Below are some screenshots of the graphical components that are showcased in the Profile Tab.



TIMELINE

Date	Description
02/04/20	Proposed project idea to Professor Wolberg and discussed the technology required. Identified scientific papers to base our initial research around.
03/06/20	Finished gathering data from Kaggle and DermNet. Stored them locally on Naman's computer.
03/10/20	Created a python resize script to make all images uniform, generated `.lst` file.
03/12/20	Experimented with some example SageMaker notebooks and understood the `image-classification` training image. Trained a model to understand everyday objects using a sample dataset.
03/19/20	Established training job for `image-classification` model using our skin conditions dataset. Deployed an endpoint hosting trained model and tested accuracy (dataset with 3 classes).
03/24/20	Discussed the option of using hyper-parameter tuning job (to achieve the most accurate trained model). Began thinking about how to approach React Native, front-end application.
04/02/20	Ran a hyper-parameter tuning job and had a barebones React Native app set up.
04/18/20	Established and "drew-out" preliminary designs and features for each of the 3 tabs (diagnosis, remedies, profile).
04/23/20	Added and finalized 3 tabs for the application, removed unreliable class from dataset (light disease had images from extremely varying angles), and populated Firebase database with remedies to display on React Native application.
04/28/20	Established table of contents for report (assigned sections to each team member). Began writing report.
05/07/20	Cleaned up remedies tab with cards for each disease, added more features to diagnosis tab and implemented cloud storage for storing diagnosis images. (helpful for building our "my diagnoses" feature on profile tab)
05/11/20	Started making presentation.
05/14/20	Presented our final presentation and project results.
05/18/20	Finished report.

CONCLUSION

The senior design project, *Skinnefy* was a collaborative effort that not only highlighted the capability of our team and the modern-day technologies that were used, but also a lot of the failures that the team came across. While we had the unique chance of exploring cloud-based Machine Learning solutions, we were also struck with the reality of paying hefty bills for, say, hyperparameter jobs.

However, the team **is** satisfied and has no regrets with the project's path/timeline as the project familiarized us with such technologies far and beyond what is the scope of the college

curriculum. Hands on experience with application development and Machine Learning architecture gave each of the team members with needed exposure and provided us with concrete expectations with regards to today's tech scene.

On the other hand, of course there were a multitude of things we could have done better. While training we noticed a huge invariance in dataset sizes ranging across the different conditions we had. Certain conditions had double the images when compared to others, and some were taken in multiple different angles and distances from the actual skin condition in question. The challenge of taking an image from *x* angle and *y* distance from the skin condition and getting accurate results every time was not sufficiently addressed. If the team were to embark on this journey again, we would consider the existence of these challenges and deal with them accordingly. The team also recognizes that other platforms, such as Google Cloud or Microsoft Azure could be used in future iterations.

All in all, we consider this a success as we were able to manifest our vision into reality.

APPENDIX

App.js

```
import React from 'react';
import { StyleSheet, View, YellowBox, StatusBar } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { SplashScreen } from 'expo';

// import BottomTabNavigator
import BottomTabNavigator from './navigation/BottomTabNavigator';

// import auth flow screens
import LoginScreen from './screens/LoginScreen';

// for authentication
import { useAuth } from './utils/auth';
import { userContext } from './utils/auth';

import {decode, encode} from 'base-64'

if (!global.btoa) {
  global.btoa = encode
}
if (!global.atob) {
  global.atob = decode
}

YellowBox.ignoreWarnings(['Setting a timer']);

const Stack = createStackNavigator();
// root stack navigator for the application

export default function App() {
```

```
  const { initializing, user } = useAuth();

  if(initializing) { return null; }
  else {
    SplashScreen.hide();
    return (
      <userContext.Provider value = {{ user }}>
        <View style={styles.container}>
          {Platform.OS === 'ios' && <StatusBar barStyle="default"/>}
          <NavigationContainer>
              {(user) ? (
                <BottomTabNavigator />
              ) : (
                <Stack.Navigator>

                  <Stack.Screen
                    name="Login"
                    component={LoginScreen}
                    options={{
                      headerShown: false,
                    }}
                  />
                </Stack.Navigator>
              )}
          </NavigationContainer>
        </View>
      </userContext.Provider>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#ffffff",
  },
});
```

TabBarIcon.js

```
import * as React from 'react';
import { Ionicons } from '@expo/vector-icons';

export default function TabBarIcon(props) {
  return (
    <Ionicons
      name={props.name}
      size={props.post ? 45 : 30}
      style={{ marginBottom: props.post ? 0 : -5 }}
      color={props.focused ? "#2f95dc" : "#ccc"}
    />
```

```
  );
}
```

BottomTabNavigator.js

```
import * as React from 'react';
import { View, StyleSheet, Text, } from 'react-native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

import DiagnosisStackNavigator from './DiagnosisStackNavigator';
import ProfileStackNavigator from './ProfileStackNavigator';
import RemediesStackNavigator from './RemediesStackNavigator';
import TabBarIcon from '../components/TabBarIcon';

const BottomTab = createBottomTabNavigator();

export default function BottomTabNavigator() {
    return (
        <BottomTab.Navigator>
            <BottomTab.Screen
                name="Remedies"
                component={RemediesStackNavigator}
                options={{
                    tabBarIcon: ({ focused }) =>
                        <TabBarIcon focused={focused} name="ios-search"
/>,
                }}
            />
            <BottomTab.Screen
                name="Diagnosis"
                component={DiagnosisStackNavigator}
                options={{
                    tabBarIcon: ({ focused }) =>
                        <TabBarIcon focused={focused} name="ios-heart-
half" />,
                }}
            />
            <BottomTab.Screen
                name="Profile"
                component={ProfileStackNavigator}
                options={{
                    tabBarIcon: ({ focused }) =>
                        <TabBarIcon focused={focused} name="ios-person"
/>,
                }}
            />
        </BottomTab.Navigator>
    )
}
```

DiagnosisStackNavigator.js

```
import * as React from 'react';
import { createStackNavigator } from '@react-navigation/stack';

import CameraScreen from '../screens/CameraScreen';
import ResultsScreen from '../screens/ResultsScreen';

const Stack = createStackNavigator();

export default function DiagnosisStackNavigator() {
    return (
        <Stack.Navigator>
            <Stack.Screen
                name='Camera'
                component={CameraScreen}
                options={{
                    headerShown: false,
                }}
            />
            <Stack.Screen
                name='Results'
                component={ResultsScreen}
                options={{
                    headerShown: true,
                    title: "Here's what we think"
                }}
            />
        </Stack.Navigator>
    )
}
```

ProfileStackNavigator.js

```
import * as React from 'react';
import { createStackNavigator } from '@react-navigation/stack';

import ProfileScreen from '../screens/ProfileScreen';
import ProfileEditScreen from '../screens/ProfileEditScreen';

const Stack = createStackNavigator();

export default function ProfileStackNavigator() {
    return (
        <Stack.Navigator>
            <Stack.Screen
                name='Profile'
                component={ProfileScreen}
                options={{
                    headerShown: true,
                }}
            />
            <Stack.Screen
```

```
                name='Edit Profile'
                component={ProfileEditScreen}
                options={{
                    headerShown: true,
                }}
            />
        </Stack.Navigator>
    )
}
```

RemediesStackNavigator.js

```
import * as React from 'react';
import { createStackNavigator } from '@react-navigation/stack';

import RemediesScreen from '../screens/RemediesScreen';
import DetailsScreen from '../screens/DetailsScreen';

const Stack = createStackNavigator();

export default function RemediesStackNavigator() {
    return (
        <Stack.Navigator>
            <Stack.Screen
                name='Conditions and Remedies'
                component={RemediesScreen}
                options={{
                    headerShown: true,
                }}
            />
            <Stack.Screen
                name='Details'
                component={DetailsScreen}
                options={{
                    headerShown: true,
                }}
            />
        </Stack.Navigator>
    )
}
```

CameraScreen.js

```
import * as React from 'react';
```

```
import { StyleSheet, TouchableOpacity, View, Text } from 'react-native';
import { Ionicons } from '@expo/vector-icons';

// for access to camera on android phone
import { Camera } from 'expo-camera';
// for using camera again if tab is in focus again
import { useIsFocused } from '@react-navigation/native';

const CAMERA_RATIO = "4:3";
var camera;

export default function CameraScreen({ navigation, route }) {
    // navigation gives access to the React Navigation method 'navigate',
    // which will allow us to jump to the results screen when the picture
is taken.
    const [hasPermission, setHasPermission] = React.useState(null);
    const [size, setSize] = React.useState(null);
    const [type, setType] = React.useState(Camera.Constants.Type.back);

    // to re-render camera component once add post tab is clicked again
after
    // being clicked away
    const isFocused = useIsFocused();

    React.useEffect(() => {
      (async () => {
        const { status } = await Camera.requestPermissionsAsync();
        setHasPermission(status === 'granted');
      })();
    }, [])

    if (hasPermission === null) {
      return <View />;
    }

    if (hasPermission === false) {
      return
      <Text>
        No access to camera
      </Text>;
    }

    const onPictureSaved = photo => {
      navigation.navigate('Results', { data: photo })
    }

    const getPictureSizes = async () => {
      if(camera) {
        const availableSizes = await
camera.getAvailablePictureSizesAsync(CAMERA_RATIO);
        setSize(availableSizes[0]);
      }
    }
```

```
    return (
      <View style={{ flex: 1, }}>
          { isFocused &&
          <Camera style={{ flex: 1, }} type={type} ratio={CAMERA_RATIO}
            autoFocus={false} pictureSize={size}
            ref={ref => { camera = ref; }} autoFocus
            onCameraReady={getPictureSizes}
          >
            <View style={{flex: 1, flexDirection: "column",
justifyContent: "flex-end"}}>
              <Text style={{fontWeight: "bold", padding: 25, fontSize: 15,
color: "#fff", alignSelf: "center",}}>
                Take a picture of your skin condition
              </Text>
              <View style={{paddingBottom: 20, flexDirection: "row",
justifyContent: "space-evenly",
                alignItems: "center",}}>
                <TouchableOpacity
                    onPress={() => {
                    camera.takePictureAsync({
                      base64: true,
                      compression: 0.0,
                      onPictureSaved: onPictureSaved});
                    }}
                >
                    <Ionicons
                      name='ios-radio-button-off'
                      color="#fefefe"
                      size={70}
                    />
                </TouchableOpacity>
                <TouchableOpacity
                    onPress={() => {
                    setType(
                    type === Camera.Constants.Type.back
                        ? Camera.Constants.Type.front
                        : Camera.Constants.Type.back
                    );
                }}>
                    <Ionicons
                    name='ios-refresh'
                    color="#fefefe"
                    size={35}
                    />
                </TouchableOpacity>
              </View>
            </View>
          </Camera>
          }
      </View>
    );
}
```

```
const styles = StyleSheet.create({
  container: {
    flexDirection: "column",
    flex: 1,
  },
  input: {
    backgroundColor: "#fefefe",
    borderRadius: 5, borderColor: "#dddddd", borderWidth: 1,
    padding: 10,
  },
  button: {
    paddingVertical: 10,
    backgroundColor: "#dddddd",
    borderColor: "#cccccc",
  },
  sliderThumb: {
    borderColor: "#bbbbbb",
    borderWidth: 1,
    elevation: 1,
  }
});

CameraScreen.navigationOptions = {
  header: null,
};
```

DetailsScreen.js

```
import * as React from 'react';
import { View, Text, StyleSheet, ActivityIndicator, ScrollView,
ImageBackground, Dimensions } from 'react-native';
import firebase from '../utils/firebaseConfig';
var db = firebase.firestore();

export default class DetailsScreen extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            isLoading: true,
            data: null,
        }
    }
    componentDidMount() {

db.collection('conditions').doc(this.props.route.params.query).get().then(
doc => {
            this.setState({
                isLoading: false,
                data: doc.data(),
            });
        })
```

```
    }

    render() {
        const { image, displayTop, description } =
this.props.route.params;
        return (
            <React.Fragment>
            { this.state.isLoading && <Loading/>}
            { !this.state.isLoading &&
            <ScrollView style={{ flex: 1 }}
contentContainerStyle={styles.container}>
                <View style={{
                    padding: 5,
                    backgroundColor: "#fff",
                    elevation: 5,
                }}>
                    <ImageBackground
                        source={{ uri: image }}
                        style={{
                            height: 200,
                            justifyContent: "flex-end",
                        }}
                    >
                        <View style={{
                                flexDirection: "row",
                                flexWrap: "nowrap",
                                justifyContent: "center"
                            }}
                        >
                            <Text style={styles.title}>
                                {displayTop}
                            </Text>
                        </View>
                    </ImageBackground>
                </View>
                <View style={{
                        paddingVertical: 10,
                    }}
                >
                    <View style={{
                            flexDirection: "row",
                            justifyContent: "space-between",
                            marginHorizontal: 10, marginBottom: 10,
                        }}
                    >
                        <View style={{
                                flex: 0.5,
                                padding: 20,
                                backgroundColor: "white",
                                borderRadius: 10, borderColor: "#ccc",
                                borderWidth: 1, marginRight: 5,
                            }}
                        >
```

```
                            <Text style={styles.sectionTitles}>
                                Prevalence
                            </Text>
                            <Text>
                                {this.state.data.prevalence ?
this.state.data.prevalence : "No information was available."}
                            </Text>
                        </View>
                        <View style={{
                                flex: 0.5,
                                padding: 20,
                                backgroundColor: "white",
                                borderRadius: 10, borderColor: "#ccc",
                                borderWidth: 1, marginLeft: 5,
                            }}
                        >
                            <Text style={styles.sectionTitles}>
                                Nearest Doctor
                            </Text>
                            <Text>
                                {this.state.data.doctor ?
this.state.data.doctor : "No information was available."}
                            </Text>
                        </View>

                    </View>
                    <View style={{
                        padding: 20, marginHorizontal: 10,
                        backgroundColor: "white",
                        borderRadius: 10, borderColor: "#ccc",
                        borderWidth: 1, marginBottom: 10,
                    }}>
                        <Text style={styles.sectionTitles}>
                            Symptoms
                        </Text>
                        <View style={{
                                marginTop: 10,
                            }}
                        >
                        {this.state.data.symptoms ?
this.state.data.symptoms.map((x, i) => {
                                return (
                                    <Text style={{
                                            marginBottom:
StyleSheet.hairlineWidth,
                                        }}
                                        key={i}
                                    >
                                        {i + 1}: {x}
                                    </Text>
                                )
                            })
                            :
```

```
                            <Text style={styles.plainText}>
                                No information was available.
                            </Text>
                        }
                        </View>
                    </View>

                    <View style={{
                        padding: 20, marginHorizontal: 10,
                        backgroundColor: "white",
                        borderRadius: 10, borderColor: "#ccc",
                        borderWidth: 1, marginBottom: 10,
                    }}>
                        <Text style={styles.sectionTitles}>
                            Home Remedies
                        </Text>
                        <View style={{
                                marginTop: 10,
                            }}
                        >
                            {this.state.data.Home_Remedies ?
this.state.data.Home_Remedies.map((x, i) => {
                                return (
                                    <Text key={i} style={{
                                            marginBottom:
StyleSheet.hairlineWidth,
                                        }}
                                        key={i}
                                    >
                                        {i + 1}: {x}
                                    </Text>
                                )
                            })
                            :
                            <Text style={styles.plainText}>
                                No information was available.
                            </Text>}
                        </View>
                    </View>

                    <View style={{
                        padding: 20, marginHorizontal: 10,
                        backgroundColor: "white",
                        borderRadius: 10, borderColor: "#ccc",
                        borderWidth: 1, marginBottom: 10,
                    }}>
                        <Text style={styles.sectionTitles}>
                            Tips
                        </Text>
                        <View style={{
                                marginTop: 10,
                            }}
                        >
```

```
                            <Text style={styles.plainText}>
                                {this.state.data.tips ?
this.state.data.tips : "No information was available."}
                            </Text>
                        </View>
                    </View>
                </View>
            </ScrollView>
            }
        </React.Fragment>
        )
    }
}

const styles = StyleSheet.create({
    container: {
        justifyContent: "center",
        backgroundColor: "#ededed",
    },
    sectionTitles: {
        fontWeight: "bold",
        fontSize: 16,
    },
    title: {
        fontSize: 30,
        margin: 10, textAlign: "center",
        paddingVertical: 5, borderRadius: 5,
        paddingHorizontal: 15, backgroundColor: "#fefefe",
        borderColor: "#ccc", borderWidth: 1, elevation: 1,
    },
    plainText: {
        textAlign: "justify"
    }
})

function Loading() {
    return (
      <View style={{
        flex: 1,
        alignItems: "center",
        justifyContent: "center",
      }}>
        <ActivityIndicator style={styles.activityIndicator} size={50}
color="#000000" />
        <Text style={{
          marginTop: 15,
          fontSize: 13,
          color: "grey"
        }}>
            Please wait while we gather your information.
        </Text>
      </View>
    )
```

```
  }
```

LoginScreen.js

```
import * as React from 'react';
import { StyleSheet, View, Text, Button, TextInput } from 'react-native';

// import sign in util function
import { loginWithPassword, } from '../utils/auth';

export default function LoginScreen({ navigation, route }) {
    // state hook to track input for email and password
    const [email, setEmail] = React.useState("");
    const [password, setPassword] = React.useState("");
    const [errMsg, setErrMsg] = React.useState(null);
    return (
        <View style={styles.loginContainer}>
            <Text style={{ color: "#555555", fontSize: 25, marginBottom:
10 }}>
                Login to Skinnefy
            </Text>
            <View style={styles.input}>
                <TextInput
                    placeholder="johndoe@gmail.com"
                    onChangeText={(text) => setEmail(text)}
                    value={email}
                />
            </View>
            <View style={styles.input}>
                <TextInput
                    placeholder='Enter your password'
                    onChangeText={(text) => setPassword(text)}
                    value={password}
                    secureTextEntry
                />
            </View>
            <View style={{minWidth: "60%", marginTop: 20}}>
                <Button
                    disabled={(email == "") || (password == "")}
                    color="#555555"
                    title="Login"
                    onPress={() => {
                        loginWithPassword(email, password)
                        .then(response => {
                            console.log("Logged in!");
                        })
                        .catch(error => { setErrMsg(error.message) });
                    }}
                />
            </View>
            <View style={{minWidth: "60%", marginTop: 10}}>
                <Button
```

```
                    color="#aaaaaa"
                    title="Create an account"
                    onPress={() =>
console.log("navigation.navigate('Signup')")}
                />
            </View>
            {
                errMsg &&
                <Text style={{color: "red", marginTop: 15, width: "65%",
                    textAlign: "center",}}>
                    {errMsg}
                </Text>
            }
        </View>
    );
    }

const styles = StyleSheet.create({
  loginContainer: {
    flex: 1,
    alignItems: "center",
    justifyContent: "center",
  },
  input: {
    elevation: 1, backgroundColor: "#e0e0e0",
    borderRadius: 10, width: "65%", marginTop: 10,
    padding: 10,
  }

});

LoginScreen.navigationOptions = {
  header: null,
};
```

ProfileDetailsScreen.js

```
import * as React from 'react';
import { View, StyleSheet, Text, } from 'react-native';
import { Badge } from 'react-native-elements';

export default function ProfileDetailsScreen({ navigation, route }) {
    return (
        <View style={{ flex: 1 }}>
            <Text style={{fontSize:15, color: "#5a5d81"}}>
                I have a history of...
            </Text>
            <View style={{
                paddingVertical: 10,
                flexDirection: "row", flexWrap: "wrap",
                justifyContent: "center"
            }}>
```

```
                {
                    data.history.map((x, i) => {
                        return (
                            <Badge
                                badgeStyle={{
                                    backgroundColor: "#dbdbdb",
                                    borderColor: "#cccccc",
                                    borderWidth: 0.5,
                                    elevation: 1,
                                    padding: 14,
                                    margin: 3,
                                }}
                                key={i}
                                value={x}
                                textStyle={{
                                    color: "#555"
                                }}
                            />
                        )
                    })
                }
            </View>
        </View>
    )
}
```

ProfileEditScreen.js

```
import * as React from 'react';
import { View, TouchableOpacity, Text, StyleSheet } from 'react-native';

export default function ProfileEditScreen({ navigation, route }) {
    return (
        <View style={styles.container}>
            <Text>
                Edit
            </Text>
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
    },
})
```

ProfileScreen.js

```
import * as React from 'react';
```

```
import { View, TouchableOpacity, Text, StyleSheet,
    ScrollView, ActivityIndicator, Dimensions, Image } from 'react-
native';

import { Ionicons } from  '@expo/vector-icons';
import { Avatar, Badge, Overlay } from 'react-native-elements'

import { logout, useSession } from '../utils/auth';
import { useFirestoreDoc, useDiagnosisOnce } from '../utils/db';

const adjustedDimension = Dimensions.get('window').width / 3;

export default function ProfileScreen({ navigation, route }) {
    navigation.setOptions({
        headerRight: () => {
            return (
                <View style={{flexDirection: "row", flexWrap: "nowrap",
marginRight: 5,}}>
                    <TouchableOpacity style={{ padding: 10}}
                        onPress={() => {
                            navigation.navigate("Edit Profile");
                        }}
                    >
                        <Ionicons
                            name="ios-build"
                            size={30}
                            color="#333"
                        />
                    </TouchableOpacity>
                    <TouchableOpacity style={{ padding: 10}}
                        onPress={logout}
                    >
                        <Ionicons
                            name="ios-log-out"
                            size={30}
                            color="#333"
                        />
                    </TouchableOpacity>
                </View>
            )
        }
    })

    const user = useSession();
    console.log(user);
    const { isLoading, data } = useFirestoreDoc('users', user.uid);

    return (
        <React.Fragment>
            { isLoading && <Loading/>}
            { !isLoading &&
            <ScrollView style={{
                    flex: 1
```

```
            }}
        >
            <View style={{
                padding: 15, backgroundColor: "#fefefe",
                elevation: 2,
            }}>
                <Avatar
                    title="L"
                    name="Lauren Smith"
                    size={120}
                    rounded
                    source={{
                        uri: data.img_src
                    }}
                    containerStyle={{
                        borderColor: "#555",
                        borderWidth: 0.5, elevation: 1,
                        alignSelf: "center",
                    }}
                />
                <View style={{
                    flexDirection: "row", flexWrap: "nowrap",
                    justifyContent: "space-evenly", paddingTop: 10,
                }}>
                    <Text style={{
                        fontSize: 25, color: "#333",
                    }}>
                        {data.info.first} {data.info.last}
                    </Text>
                    <View style={{
                        backgroundColor: "pink",
                        borderRadius: 5, paddingHorizontal: 10,
                        paddingVertical: 2.5,
                    }}>
                        <Text style={{
                            fontSize: 20,
                        }}>
                            {data.info.age}
                        </Text>
                    </View>
                </View>
            </View>
            <View style={{
                padding: 20,
                flex: 1,
            }}>
                <View>
                    <Text style={{fontSize:15, color: "#5a5d81"}}>
                        I have a history of...
                    </Text>
                    <View style={{
                        paddingVertical: 10,
                        flexDirection: "row", flexWrap: "wrap",
```

```
                        justifyContent: "center"
                    }}>
                        {
                            data.history.map((x, i) => {
                                return (
                                    <Badge
                                        badgeStyle={{
                                        backgroundColor: "#dbdbdb",
                                        borderColor: "#cccccc",
                                        borderWidth: 0.5,
                                        elevation: 1,
                                        padding: 14,
                                        margin: 3,
                                        }}
                                        key={i}
                                        value={x}
                                        textStyle={{
                                            color: "#555"
                                        }}
                                    />
                                )
                            })
                        }
                    </View>
                </View>
                <View style={{
                    marginVertical: 10,
                    flex: 1,
                }}>
                    <Text style={{fontSize:15, color: "#5a5d81"}}>
                        My Diagnoses
                    </Text>
                    {data.diagnoses.length == 0 &&
                        <View style={{
                            justifyContent: "center",
                            alignItems: "center",
                            flex: 1,
                        }}>
                            <Text style={{
                                color: "#555",
                                fontSize: 18,
                            }}>
                                You have no Self-Diagnoses
                            </Text>
                            <Text style={{
                                color: "#999",
                                fontSize: 11,
                            }}>
                                Press on the Diagnosis tab to make
one!
                            </Text>
                        </View>
                    }
```

```
                        {data.diagnoses.length > 0 &&
                            <View style={{
                                padding: 10, flexWrap: "wrap",
flexDirection: "row",
                                justifyContent: "center",
                            }}>
                                {data.diagnoses.map((diagnosis, key) => {
                                    return (
                                        <Diagnosis id={diagnosis}
key={key} />
                                    )
                                })}
                            </View>
                        }
                    </View>
                </View>

            </ScrollView>
            }
        </React.Fragment>
    )
}

export function Diagnosis(props) {
    const [modal, setModal] = React.useState(false);
    const { isLoading, data } = useDiagnosisOnce(props.id);
    return (
        <React.Fragment>
        { isLoading && null }
        {
            !isLoading &&
            <View>
                <Overlay
                    isVisible={modal}
                    onBackdropPress={() => {
                        setModal(false);
                    }}
                    overlayStyle={{
                        padding: 0, height: 450, elevation: 0,
                        width: Dimensions.get('window').width,
                        backgroundColor: "transparent",
                    }}
                >
                    <View style={styles.centeredView}>
                        <Image
                            source={{ uri: data.image }}
                            style={{
                                width: Dimensions.get('window').width -
30,
                                height: Dimensions.get('window').width -
30,
                            }}
```

```
                        />
                    </View>
                </Overlay>
                <TouchableOpacity
                    style={{ margin: 5, }}
                    onPress={() => {
                        setModal(true);
                    }}
                >
                    <Image
                        source={{ uri: data.image }}
                        style={{
                            width: adjustedDimension,
                            height: adjustedDimension,
                            borderRadius: 10,
                        }}
                    />
                </TouchableOpacity>
            </View>
        }
        </React.Fragment>
    )
}

function Loading() {
  return (
    <View style={{
      flex: 1,
      alignItems: "center",
      justifyContent: "center",
    }}>
      <ActivityIndicator style={styles.activityIndicator} size={50}
color="#000000" />
      <Text style={{
        marginTop: 15,
        fontSize: 13,
        color: "grey"
      }}> Please wait while we get your data </Text>
    </View>
  )
}
const styles = StyleSheet.create({
    button: {
        backgroundColor: "#ddd",
        borderColor: "#aaa", borderWidth: 0.5,
        elevation: 0.5, padding: 10, borderRadius: 5,
    },
    centeredView: {
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
    },
})
```

RemediesScreen.js

```
import * as React from 'react';
import { StyleSheet, ScrollView } from 'react-native';
import { Card, CardTitle, CardContent, CardAction, CardButton, CardImage }
from 'react-native-cards';

export default class RemediesScreen extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return (
            <React.Fragment>
            <ScrollView style={{ flex: 1 }}
contentContainerStyle={styles.container}>
                {prepared_conditions.map((condition, key) => {
                    return (
                        <Card style={{
                            backgroundColor: "#efefef",
                            borderRadius: 10, elevation: 2,
                        }}
                        key={key}>
                            <CardImage
                                source={{uri : condition.image}}
                                style={{
                                    height: 150,
                                }}
                            />
                            <CardTitle title={condition.name} />
                            <CardContent text={condition.description} />
                            <CardAction separator={true} inColumn={false}>
                                <CardButton
                                    onPress={() =>
{this.props.navigation.navigate('Details', {
                                        query: condition.ref,
                                        displayTop: condition.name,
                                        description:
condition.description,
                                        image: condition.image,
                                    })}}
                                    title="Details"
                                    color="#FEB557"
                                />
                            </CardAction>
                        </Card>
                    )
                })
                }
            </ScrollView>
```

```
        </React.Fragment>
        )
    }
}

const prepared_conditions = [
    {
        name: "Acne",
        image:
"https://assets.nhs.uk/prod/images/S_0917_acne_M1080444.width-
320_4r0CUty.jpg",
        description: "Acne is a skin condition that occurs when your hair
follicles become plugged with oil and dead skin cells.\
         It often causes whiteheads, blackheads or pimples, and usually
appears on the face, forehead, chest, upper back and shoulders.",
        ref: "acne",
    },
    {
        name: "Eczema",
        image: "https://i0.wp.com/cdn-
prod.medicalnewstoday.com/content/images/articles/014/14417/eczema-can-
cause-dry-and-itchy-rashes-image-credit-g-steph-rocket-
2015.jpg?w=1155&h=1654",
        description: "Eczema is a condition where patches of skin become
inflamed, itchy, red, cracked, and rough. Blisters may sometimes occur.",
        ref: "eczema",
    },
    {
        name: "Melanoma",
        image: "https://www.news-
medical.net/image.axd?picture=2019%2F12%2F%40shutterstock_1391821073.jpg",
        description: "Melanoma is a serious form of skin cancer that
begins in cells known as melanocytes.",
        ref: "melanoma",
    },
    {
        name: "Nail Fungus",
        image:
"https://img.webmd.com/dtmcms/live/webmd/consumer_assets/site_images/artic
les/health_tools/toenail_fungus_slideshow/princ_rm_photo_of_toenail_fungus
.jpg",
        description: "Nail fungus is a common condition that begins as a
white or yellow spot under the tip of your fingernail or toenail. As the
fungal infection goes deeper, nail fungus may cause your nail to discolor,
thicken and crumble at the edge.",
        ref: "nail_fungus",
    },
    {
        name: "Psoriasis",
        image:
"https://images.medicinenet.com/images/image_collection/pediatrics/psorias
is-9-9.jpg",
```

```
        description: "Psoriasis is a skin disorder that causes skin cells
to multiply up to 10 times faster than normal. This makes the skin build
up into bumpy red patches covered with white scales.",
        ref: "psoriasis",
    },
    {
        name: "Scabies",
        image: "https://images.medicinenet.com/images/slideshow/stds-s3-
photo-of-scabies.jpg",
        description: "Scabies, also known as the seven-year itch, is a
contagious skin infestation by the mite Sarcoptes scabiei. The most common
symptoms are severe itchiness and a pimple-like rash.",
        ref: "scabies",
    },
    {
        name: "Seborrheic Keratoses",
        image: "https://i0.wp.com/cdn-
prod.medicalnewstoday.com/content/images/articles/320/320742/seborrheic-
keratosis.jpg?w=1155&h=1541",
        description: "Seborrheic keratosis is a common, harmless,
noncancerous growth on the skin. It usually appears as a pale, black, or
brown growth on the back, shoulders, chest, or face.",
        ref: "seborrheic_keratoses",
    },
    {
        name: "Tinea Ringworm",
        image:
"https://img.webmd.com/dtmcms/live/webmd/consumer_assets/site_images/artic
le_thumbnails/slideshows/ringworm_slideshow/650x350_ringworm_slideshow.jpg
",
        description: "Ringworm is a common fungal skin infection otherwise
known as tinea. Ringworm most commonly affects the skin on the body (tinea
corporis), the scalp (tinea capitis), the feet (tinea pedis, or athlete's
foot), or the groin (tinea cruris, or jock itch).",
        ref: "tinea",
    },
    {
        name: "Warts",
        image:
"https://img.medscape.com/thumbnail_library/ps_200115_warts_hands_800x450.
jpg",
        description: "A wart is a small growth with a rough texture that
can appear anywhere on the body. It can look like a solid blister or a
small cauliflower. Warts are caused by viruses in the human papillomavirus
(HPV) family.",
        ref: "wart",
    }
]

const styles = StyleSheet.create({
    container: {
        justifyContent: "center",
        padding: 20,
```

```
        paddingTop: 30,
        backgroundColor: "#fefefe"
    },
})
```

ResultsScreen.js

```
import * as React from 'react';
import { Image, Text, StyleSheet, View, ScrollView, TouchableOpacity,
ActivityIndicator, Dimensions } from 'react-native';
import Ionicons from '@expo/vector-icons';
import { BarChart, } from 'react-native-chart-kit';

import { useEndpoint } from '../utils/sagemakerCalls';
import { uploadDiagnosisSequence } from '../utils/storage';
import { useSession } from '../utils/auth';

export default function ResultsScreen({ navigation, route }) {
    const user  = useSession();
    const { isLoading, data, err } = useEndpoint(route.params.data);

    return (
        <ScrollView style={styles.container}>
            <View style={{padding: 10, flexDirection: "row",
justifyContent: "center",}}>
                <Image
                style={{ width: 150, height: 200, borderRadius: 10,
borderColor: "#333333", borderWidth: 2, }}
                source={
                    { uri: route.params.data.uri }
                }
                />
            </View>
            <View style={styles.resultsContainer}>
                <Text style={{color: "#666666", fontSize: 14,
marginBottom: 20,}}>
                    According to our AI powered prediction engine these
are your results
                </Text>
                {
                    isLoading &&
                    <View style={{flex: 1, justifyContent: "center",
alignItems: "center"}}>
                        <ActivityIndicator
style={styles.activityIndicator} size={50} color="#000000" />
                    </View>
                }

                {
                    !isLoading &&
                    <View style={{flex: 1, justifyContent: "center",
alignItems: "center",}}>
```

```
                        {err ?
                            <Text style={{color: "red"}}>ERROR:
{err}</Text>
                        :
                            <React.Fragment>
                                <BarChart
                                    // style={graphStyle}
                                    fromZero={true}
                                    segments={4}
                                    withInnerLines={false}
                                    data={{
                                        labels: data.labels.slice(0, 2),
                                        datasets: [
                                            {
                                                data:
data.dataset.slice(0, 2).map(val => {
                                                    return val * 100;
                                                }),
                                            },
                                        ],
                                    }}
                                    yAxisSuffix="%"
                                    width={Dimensions.get('window').width
- 20}
                                    height={200}
                                    chartConfig={{
                                        backgroundColor: '#1cc910',
                                        backgroundGradientFrom: '#fff',
                                        backgroundGradientTo: '#999',
                                        decimalPlaces: 2, // optional,
defaults to 2dp
                                        color: (opacity = 255) => `rgba(0,
0, 0, ${opacity})`,
                                        style: {
                                        borderRadius: 16
                                        },

                                    }}
                                    style={{
                                        borderRadius: 5,
                                        padding: 5,
                                        backgroundColor: "#999",
                                        elevation: 5,
                                    }}
                                />
                                <View style={{ flexDirection: "row",
flexWrap: "nowrap", marginTop: 20, justifyContent: "flex-end"}}>
                                    <TouchableOpacity style={{
                                            backgroundColor: "#ededed",
                                            borderColor: "ccc",
borderWidth: 0.5,
                                            elevation: 1,
paddingHorizontal: 15,
```

```
                                            paddingVertical: 5,
borderRadius: 5,
                                        }} onPress={() => {

uploadDiagnosisSequence(user.uid, {
                                                data: data,
                                                title: data.labels[0],
                                            }, route.params.data);
                                            navigation.navigate("Camera");
                                        }}
                                    >
                                        <Text style={{
                                                fontSize: 15,
                                            }}
                                        >
                                            Save
                                        </Text>
                                    </TouchableOpacity>
                                </View>
                            </React.Fragment>
                        }
                    </View>

                }

            </View>
        </ScrollView>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
    },
    resultsContainer: {
        flex: 1,
        padding: 20,
        backgroundColor: "#fefefe",

    }
})
```

auth.js

```
import * as React from 'react';
import firebase from './firebaseConfig';

var auth = firebase.auth();

export const userContext = React.createContext({
    user: null,
});
```

```
export const useSession = () => {
  const { user } = React.useContext(userContext);
  return user;
}

export const useAuth = () => {
    const [state, setState] = React.useState(() => {
        const user = firebase.auth().currentUser;
        return { initializing: !user, user, }
    })
    function onChange(user) {
        setState({ initializing: false, user });
    }
    React.useEffect(() => {
        // listen for auth changes
        const unsubscribe = firebase.auth().onAuthStateChanged(onChange);
        // unsubscribe to the listener when unmounting
        return () => unsubscribe();
    }, []);
    return state;
}

export function logout() {
    auth.signOut().then(() => console.log("User Signed out"))
    .catch(() => console.log(error.message));
}

export function loginWithPassword(email, password) {
    return auth.signInWithEmailAndPassword(email, password);
}
```

db.js

```
import * as React from 'react';
import firebase from './firebaseConfig';
import { TouchableOpacity, ActivityIndicator } from 'react-native';
import { Ionicons } from '@expo/vector-icons';
var db = firebase.firestore();

// Functions for pulling database datat, filtering and submitting custom
queries

export const useFirestoreDoc = (collectionName, docName) => {
    const [docState, setDocState] = React.useState({ isLoading: true,
data: null });
    React.useEffect(() => {
        return db.collection(collectionName)
            .doc(docName)
            .onSnapshot(doc => {
            setDocState({ isLoading: false, data: doc.data() });
        });
```

```
    }, []);
    return docState;
}

export const useDiagnosisOnce = (postDocName) => {
    const [docState, setDocState] = React.useState({ isLoading: true,
data: null });
    const compileData = async () => {
        var diagnosisDoc = await
db.collection("diagnoses").doc(postDocName).get();
        setDocState({
            isLoading: false,
            data: diagnosisDoc.data(),
        });
    }
    React.useEffect(() => {
        compileData();
    }, []);
    return docState;
}
```

sagemakerCalls.js

```
import * as React from 'react';
var AWS = require('aws-sdk/dist/aws-sdk-react-native');
var credentials = require("./awsConfig.json");

var sagemakerruntime = new AWS.SageMakerRuntime({accessKeyId:
credentials.accessKeyId,
    secretAccessKey: credentials.secretAccessKey, region: "us-east-2",});

const endpoint_name = "endpoint-5-6-2020";
const conditions = ["Acne", "Eczema", "Melanoma", "Nail Fungus",
                    "Psoriasis", "Scabies", "Seborrheic Keratoses",
"Ringworm", "Warts"];

FileReader.prototype.readAsArrayBuffer = function (blob) {
    if (this.readyState === this.LOADING) throw new
Error("InvalidStateError");
    this._setReadyState(this.LOADING);
    this._result = null;
    this._error = null;
    const fr = new FileReader();
    fr.onloadend = () => {
        if(!fr.result) {
            throw new Error("Image too big!")
        }
        const content = atob(fr.result.substr("data:application/octet-
stream;base64,".length));
        const buffer = new ArrayBuffer(content.length);
        const view = new Uint8Array(buffer);
        view.set(Array.from(content).map(c => c.charCodeAt(0)));
```

```
        this._result = buffer;
        this._setReadyState(this.DONE);
    };
    fr.readAsDataURL(blob);
}

export function useEndpoint(image) {
    const [queryState, setQueryState] = React.useState({ isLoading: true,
data: null, err: null });

    const sendAndRetrieve = async () => {
        try {
            const response = await fetch(image.uri);
            const blob = await response.blob();
            sagemakerruntime.invokeEndpoint({
                Body: blob,
                EndpointName: endpoint_name,
                ContentType: "application/x-image",
            }, function(err, data) {
                if(err) {
                    setQueryState({
                        isLoading: false,
                        data: null,
                        err: err.message
                    })
                }
                else {
                    let prediction = prepareResponse(data.Body);
                    setQueryState({
                        isLoading: false,
                        data: prediction,
                        err: null
                    })
                }
            });
        }
        catch (error) {
            console.log(error)
        }
    }
    // // convert the base64 encoding string to uint8 typearray
    // var result = convertToArray(image.base64);
    React.useEffect(() => {
        sendAndRetrieve();
    }, []);
    return queryState;
}

function prepareResponse(responseBuffer) {
    var i, j;
    const predictionValues = JSON.parse(responseBuffer.toString());
    console.log(predictionValues)
    // manually sort two arrays. conditions based on predictions
```

```
    for(i = 0; i < predictionValues.length; i++) {
        for(j = i; j < predictionValues.length; j++) {
            if(predictionValues[i] < predictionValues[j]) {
                // swap first array
                let tmp = predictionValues[i];
                predictionValues[i] = predictionValues[j];
                predictionValues[j] = tmp;
                // swap second array
                tmp = conditions[i];
                conditions[i] = conditions[j];
                conditions[j] = tmp;
            }
        }
    }
    console.log(predictionValues);
    const prediction = {
        labels: conditions,
        dataset: predictionValues,
    }
    return prediction;
}
```

storage.js

```
import * as React from 'react';
import firebase from './firebaseConfig';

var storage = firebase.storage();

// for most functions we also need access to the db API
var db = firebase.firestore();

export async function uploadDiagnosisSequence(user, payload, image) {
    /* require the user so that the image is stored correctly in
    the storage bucket. the title is used to name the file, while image
    is a json with a key for the base64 encoded string */
    let suffix = Math.round(Math.random() * 10000);
    const path = "diagnoses/" + user + "/" +
payload.title.toLowerCase().replace(/ /g, "_") + "_" + suffix.toString() +
".jpg";
    var storageRef = storage.ref();

    const response = await fetch(image.uri);
    const blob = await response.blob();

    var uploadTask = storageRef.child(path).put(blob);

    const saveToDatabase = async () => {
        const timestamp = firebase.firestore.FieldValue.serverTimestamp();
        try {
            const url = await uploadTask.snapshot.ref.getDownloadURL();
            const diagnosisDocRef = await db.collection("diagnoses").add({
```

```
                user: user,
                image: url,
                created: timestamp,
            });
            const userDocRef = await
db.collection("users").doc(user).update({
                diagnoses:
firebase.firestore.FieldValue.arrayUnion(diagnosisDocRef.id),
            })
        } catch (err) { console.log(err); }
        finally {
            console.log("Completed this.")
        }
    }
    uploadTask.on(firebase.storage.TaskEvent.STATE_CHANGED, // or
'state_changed'
        function(snapshot) {
            // Get task progress, including the number of bytes uploaded
and the total number of bytes to be uploaded
            var progress = (snapshot.bytesTransferred /
snapshot.totalBytes) * 100;
            console.log('Upload is ' + progress + '% done');
            switch (snapshot.state) {
                case firebase.storage.TaskState.PAUSED: // or 'paused'
                    console.log('Upload is paused');
                    break;
                case firebase.storage.TaskState.RUNNING: // or 'running'
                    console.log('Upload is running');
                    break;
            }
        }, function(error) {
            // A full list of error codes is available at
            // https://firebase.google.com/docs/storage/web/handle-errors
            console.log(error.code);
        }, saveToDatabase
    );
}
```

resize.py

```
from PIL import Image
import os, sys

path =
"/Users/siddharthrajan/Users/siddharthrajan/dataset/validation/wart_viral_
infections/"
dirs = os.listdir( path )

def resize():
    print("Resizing...")
    for item in dirs:
        if os.path.isfile(path+item):
```

```
            print("Processing: " + item)
            im = Image.open(path+item)
            imResize = im.resize((244,244), Image.ANTIALIAS)

imResize.save("/Users/siddharthrajan/dataset/validation/wart_viral_infecti
ons/" + item, 'JPEG', quality=90)
    print("Done.")

resize()
```

scraper.py

```
"""Scrapes dermnet for all images.
"""
import os
import re
import urllib.request
from urllib.parse import urlparse
from urllib.parse import urljoin
import pickle
from bs4 import BeautifulSoup

DERMNET_PIC_PAGE = "http://www.dermnet.com/dermatology-pictures-skin-
disease-pictures/"
DERMNET_HOME_PAGE = "http://www.dermnet.com/"

def genClass2URL():
    """Create a dictionary from each DermNet class to a URL.

    @return image_dict: dictionary containing image urls for 23 skin
disease classes.
    """

    # open DermNet root directory and get class links
    soup = soupify(DERMNET_PIC_PAGE)
    class_links = soup.find("table").find_all("a")
    n_links = len(class_links)
    print("Found {} total image classes.".format(n_links))
    n_total = 0

    img_dict = {}
    for i, link in enumerate(class_links):
        abs_link = urljoin(DERMNET_HOME_PAGE, link.get('href'))
        class_name = re.sub(r'[^a-z0-9A-Z\s]+', '', link.string)
        print('\nFetching URLs for class [{}/{}]: {}'.format(i + 1,
n_links, class_name))
```

```
        # add to final dictionary {class_name: list of image links}
        class_images = genClassImages(abs_link)
        n_total += len(class_images)
        print('Fetched {} images. Total of {}
images.'.format(len(class_images), n_total))
        img_dict[class_name] = class_images

    return img_dict

def genClassImages(class_url):
    """Fetch list of class images
    @arg class_url: web url
    @returns class_images: list of images
    """
    images = []
    urls = genClassCategories(class_url)
    print('- Found {} total sub-classes for class.'.format(len(urls)))

    for i, url in enumerate(urls):
        print('-- Fetching images from sub-class [{}/{}]'.format(i + 1,
len(urls)))
        images.extend(genCategoryImages(url))

    return images

def genClassCategories(class_url):
    """Fetch list of categories for a single class
    @arg class_url: web url
    @returns categories: list of categories
    """
    soup = soupify(class_url)
    links = soup.find("table").find_all("a")

    categories = []
    for link in links:
        abs_link = urljoin(DERMNET_HOME_PAGE, link.get('href'))
        categories.append(abs_link)
    return categories

def genCategoryImages(cat_url):
    """Fetches all category image urls within a series of paginated links.

    @arg url: a category web address.
    @return images: A list containing image urls.
    """
    images = []
    genPageImages(cat_url, images)

    thumb_urls = genCategoryLinks(cat_url)
    # more pages in category, add images from those thumbnail pages
    for page in thumb_urls:
        genPageImages(page, images)
```

```
    return images

def genCategoryLinks(url):
    """Returns paginated links associated to a category, if any.

    @url: a category web address.
    @returns thumb_urls: A list of paginated link addresses.
    """
    soup = soupify(url)
    pages = soup.find("div", "pagination")
    thumb_urls = []

    if pages:  #there are multiple pages for this category
        for page in pages:
            if page.name == 'a' and page.string != 'Next':
                page_url = urljoin(DERMNET_HOME_PAGE, page['href'])
                thumb_urls.append(page_url)

    return thumb_urls

def genPageImages(url, image_list):
    """Finds all image links in a webpage and adds them to the image list.

    @arg url: web url; str
    @arg image_list: a list of image urls.
                     this will be modified in place.
    @return None
    """
    soup = soupify(url)
    thumbnails = soup.find_all("div","thumbnails")
    if thumbnails: ## there are thumbnails actually on the page
        for thumb in thumbnails:
            thumb_link = thumb.img['src']
            #use full image link instead of thumbnail link
            image_link = re.sub(r'Thumb',"",thumb_link)
            image_list.append(image_link)

def soupify(url):
    """Call BeautifulSoup on a webpage
    @arg url: web url; str
    @return soup: BeautifulSoup instane
    """
    html = urllib.request.urlopen(url)
    soup = BeautifulSoup(html, "lxml")
    return soup

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('out_folder', type=str, help='where to store
scraped images.')
    parser.add_argument('--dictionary', type=str, help='class2url
dictionary path')
```

```
    args = parser.parse_args()

    print('Scraping DermNet for URLs.')
    if args.dictionary:
        with open(args.dictionary, 'rb') as fp:
            image_dict = pickle.load(fp)
    else:
        image_dict = genClass2URL()
    print(image_dict)

    n_images = 0
    for klass, images in image_dict.iteritems():
        n_images += len(images)

    n_downloaded = 0

    with open(os.path.join(args.out_folder, 'backup.pkl'),'wb') as fp:
        pickle.dump(image_dict, fp)
    print('Dumped dictionary of URLs to current directory.')

    # we will now download each image
    for klass, images in image_dict.iteritems():
        # create class folders, if it doesn't exist
        class_path = os.path.join(args.out_folder, klass)
        if not os.path.exists(class_path):
            os.mkdir(class_path)

        for image in images:
            image_name = os.path.basename(image)
            file_name = os.path.join(class_path, image_name)
            # download image
            try:
                f = urllib.urlopen(image).read()
                open(file_name, 'wb').write(f)
                n_downloaded += 1
                print('Downloaded [{}/{}] images.'.format(n_downloaded,
n_images))
            except urllib.HTTPError:
                continue
```