Bizzy Moore
Professor Bridgeman
Professor Dumitriu
CS Independent Study
Spring 2020

Classification of Harmful Algal Blooms in the Finger Lakes through Machine Learning

Introduction

Harmful Algal Blooms (HABs) are a critical problem across the United States that have been negatively affecting bodies of water all across the nation. HABs take place "when colonies of algae - simple plants that live in the sea and freshwater - grow out of control and produce toxic or harmful effects on people, fish, shellfish, marine mammals and birds" [1]. It has been found that "human illnesses caused by HABs, though rare, can be debilitating or even fatal" [1]. Due to the many negative consequences of HABs, there are many scientists that are currently conducting research on their behaviors and trends. However, the transient nature of HABs in both space and time result in monitoring challenges, and it is not understood yet what triggers the blooms.

Scientists at Hobart and William Smith Colleges and the Finger Lakes Institute in Geneva, NY are part of this group of researchers who are trying to get a better understanding of the characteristics of HABs. The Finger Lakes Region has been greatly affected by HABs in recent years. Because of the many unknowns of HABs, these scientists have been using a variety of techniques in order to collect data and track the HABs. One of these methods includes taking high quality images and videos of the Finger Lakes. This remote sensing data is captured at the same time and the same location as the *in situ* water quality measurements in order to detect the HABs and their concentration in the water. These images and measurements have been taken on a consistent basis throughout the same selected locations throughout the Finger Lakes Region. This allows the lifespan of HABs in the Finger Lakes to be documented and tracked.

In order to analyze all of the data that is being collected, professors and research students are currently processing each image manually using image processing software such as Adobe Photoshop and ImageJ. After a new set of images and water quality data gets collected, they sort through the images, looking for blue-green algae, which signifies that a HAB took place. However, this process is very time consuming and tedious. This is because of the fairly large size of each data set that is collected. Therefore, manually sorting and processing each image is not efficient.

There are a few potential applications that could be created at HWS that have the ability to help the professors and research students optimize their process of analyzing HABs. The simplest application would be creating a program that takes a set of images that were taken of the Finger Lakes and sorts them based on their characteristics. A second application would take a video and identify what frames contain HABs. Another application would consist of being able to isolate or find the individual images in the image that contain the specific green color that makes up the HABs. Once the program determines where the blooms are located in the image, a false map could be created in order to clearly portray the placement of the HABs. A false map is when a specific pixel color is replaced with another pixel color in order to make the desired locations more visible. This map would allow researchers and scientists to better monitor and

study patterns and tendencies of the HABs. They would be able to track their movement where they show up. The last application would perform real-time detection of HABs from drone flights. Researchers at HWS and the Finger Lakes Institute often fly drones above the Finger Lakes and take pictures and videos with the drone's camera. It would be very convenient if it would be possible for the detection of the HABs to happen in real-time as the drone was flying over the lakes. This would give the researchers live feedback, and they would not have to wait until they got back to the lab and analyzed the data to know if HABs occurred that day. All four of these applications would optimize the amount of time that scientists could spend monitoring and analyzing the trends and patterns of HABs. They would not have to first spend hours sorting through all of the images, videos, and data. The applications would do this for them.

The four applications above can be created through the technique of machine learning. Each application differs in the degrees of difficulty it would take to complete the project. This paper discusses the process of designing and constructing the first application suggested: a program that takes a set of images that were taken of the Finger Lakes and sorts them into separate folders based on the images' characteristics.

Background

Machine Learning Overview

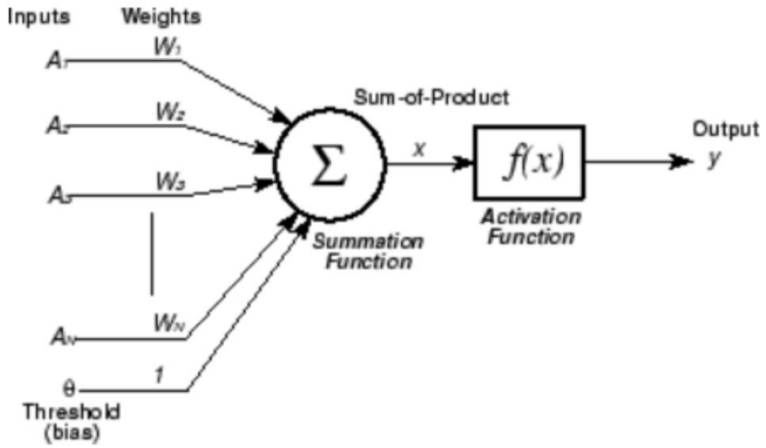
Arthur Samuel, a pioneer in the field of artificial intelligence, described machine learning as "a field of study that gives computers the ability to learn without being explicitly programmed" [2]. Machine learning has many applications. Some problems where machine learning is used to help create a solution are when the designer "can't anticipate all possible situations the agent might be in," when the designer "can't anticipate all changes over time," and when the designer does not "know how to program the solution" [2]. A machine learning model has the ability to learn many things depending on its specific architecture. A few of these possibilities include "a direct mapping from state to action," the "quality/value of actions or states," the "information about the results of possible actions the agent can take," the "classes of states (goals) whose achievement maximizes the agent's utility," and a "means to infer properties of the world from the percepts" [2]. This last type of learning is what was used for this particular project. Our model was used to learn and infer the properties that make up the HABs that occur in the Finger Lakes.

There are a variety of learning strategies that can be used by a machine learning model in order for it to receive its desired results. These include rote learning, supervised learning, reinforcement learning, and unsupervised learning. Supervised learning is the learning strategy that was used to train the model for this project. This strategy consists of providing the model with "examples of input/output pairs," and then it "must learn a function to map the inputs to outputs" [2]. The function that is created must "cover all possibilities, allowing generalization to new situations" [2]. The outputs for the pairs can be in many different forms. The outputs for our model were classification labels. These labels become very important when the model is trained and evaluated.

Artificial Neural Network

In order to implement supervised learning, an artificial neural network is used. Artificial neural networks are "a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns" [3]. The human brain is made up of "lots of tiny units called

neurons" [2]. Every neuron "is connected to many other neurons" and each one "send[s] and receive[s] electrochemical signals" [2]. A specific neuron fires "if the combination of its inputs exceeds some threshold" [2]. In an artificial neural network, these neurons are represented by artificial neurons called perceptrons. Below is an example of a perceptron. A perceptron takes "many inputs, each with a weight *w*" [2]. A positive weight has a stimulative effect on the perceptron's activation, while a negative weight has a restrictive effect. A perceptron also "may



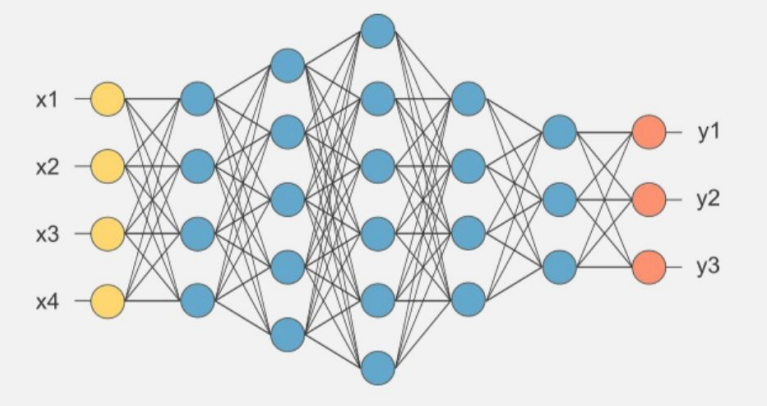
include a [threshold] bias θ, with weight 1" [2]. The perceptron calculates "the dot product between the inputs and the weights, computes a weighted sum, and then applies a step function to determine the output class label" [4]. The step function, also referred to as the activation function, determines if the perceptron gets activated or not. It is portrayed below. If the weighted sum, *x*, is greater than 0, then the perceptron exceeds the threshold and can "fire". Therefore, the activation function is set to 1 and the perceptron gets activated. If the weighted sum is less than or equal to 0, then the activation function is set to 0, and the perceptron is not activated.

$$x = \sum x_i w_i + \theta$$

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The act of "adjusting the weights and bias changes the decisions made by the neuron" [2]. Therefore, training a perceptron consists of giving it inputs and adjusting and updating the weights in order to get the desired result.

In order to deal with complicated tasks, more than one neuron or perceptron is needed. A neural network is a network of neurons. A neural network is formed when multiple "individual neurons are connected" [2]. A neural network is depicted below. The number of inputs "is determined by the pattern or input the network is to process," while the number of outputs "is determined by the result you want" [2]. The "learning goal is to learn the weights for



each neuron's inputs so that the right combination of outputs is produced for each input" [2].

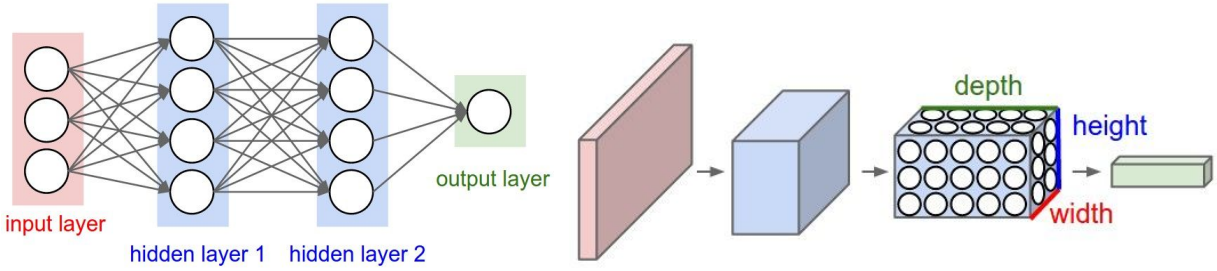
Neural networks can vary in the amount of layers that they contain. For example, a perceptron network is a "single-layer neural network" where "all inputs are connected directly to outputs" [2]. An example of this would be if in the picture above, all of the yellow inputs were directly connected to the orange outputs, and there were no blue layers in between. It is also possible for neural networks to contain extra additional layers, called hidden layers. A neural network that contains one or more hidden layers is referred to as a deep neural network and conducts deep learning [4]. The picture above is a deep neural network because it contains five hidden layers, depicted in blue.

There is no set rule on how many hidden layers should be in a neural network. However, there is usually at least one because "perceptron networks are too simple for many applications" [2]. Therefore, it is common for a network to contain at least two or more hidden layers. However, even though adding more hidden layers "add[s] additional power," it "also slows down the learning" [2]. There is also no regulation for how many neurons should be included in each layer. You have to try multiple combinations in order to find the right balance. For example, if the layers contain "too few" neurons, then the "function learned is too much of an approximation" [2]. But, if the layers contain "too many" neurons, then the "network learns examples too precisely and loses the ability to generalize" [2]. We want something in between. This is often found through experiments and logical trial and error.

There are a variety of different types of neural networks. The architecture, meaning the network's weights and the amount of layers and neurons that the network contains, varies depending on what the network is trying to classify. For this project, we used a Convolutional Neural Network (CNN). A CNN is a "special type of neural network that lends [itself] well to image understanding tasks" [4]. Since we want our network to classify images, we decided a CNN would be a good option to use for our model.

The main difference between regular neural networks and convolutional neural networks are that the layers of a CNN are "organized in 3 dimensions: width, height, and depth" [5]. The diagrams below show a regular neural network (left) and a convolutional neural network (right). The input layer for the CNN is an image. Therefore, the "width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels)" [5]. The final output of a CNN is "reduced to a single vector of probability scores" [5]. CNNs are

split up into two components: hidden layers and classification. In the hidden layers, “the network will perform a series of convolutions … during which the features are detected” [5]. A



convolution is “the mathematical combination of two functions to produce a third function” [5]. The classification section is where the model assigns the “probability for the object on the image being what the algorithm predicts it is” [5].

Training

Once the layers and the weights of the neural network model are specified, it is time to train the model. Training a model means “learning (determining) good values for all the weights and the bias from labeled examples” [6]. For supervised learning, “a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss” [6]. Loss “is a number [that indicates] how bad the model’s prediction was on a single example. If the model’s prediction is perfect, the loss is zero; otherwise, the loss is greater” [6]. Therefore, the “goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples” [6].

Labels are important in the training process because when the model is looking at the characteristics of a certain image, it needs to know which classification group the image belongs to. This allows the model to make connections between images that belong in the same group.

Before training begins, the data that is imported into the model is divided up into a training set and a test set. The test set consists of a group of images that are set aside and are only used to test the neural network once it is finished training. The data is split up into these two groups because we want to make sure that when we test the network it is not seeing any images that were used to create its classification function during its training. If pictures were repeated from the training set, then the program would have had a bias towards these images. This would not allow us to accurately evaluate how well the network was actually classifying the images.

The size of the training set and the test set depends on how large the model’s data set is. It is common for data sets to consist of thousands of data points. In this case, often 25% of the data is set aside for the test set, while the other 75% is used for training. This is because with thousands of data points, there is enough data to take a quarter of the points and still have enough diversity in the training set to accurately train the model. However, for smaller data sets that only consist of a couple hundred data points, taking 25% of the data for testing could mean sacrificing data that the model could need to increase its accuracy. This is due to there being less data to train off of, which decreases the amount of variation that the model is exposed to during training. With smaller data sets, it is common for the test set to be 10% of the original data set and the training set to be 90%. This allows for the model to train on the biggest amount of data as possible, while still leaving some data for testing.

Testing

After the model is trained, it gets tested with new data that it has not seen before. This data is from the test set that is described above. However, we do not want to test our network once, and trust only those results. This is because one test set most likely does not fully represent the data set. Therefore, the model could have had good results for those specific test images, but if you run the program again with a different test set, the results will differ depending on the input images. A technique called cross validation is commonly used during the testing stage in order to deal with this problem. The "simplest kind of cross validation" is the "holdout method" [7]. This is when the data set "is separated into two sets, called the training set and the testing set" [7]. This is similar to what was discussed above. A method that is more robust than the holdout method is k-fold cross validation. In this method, "the data set is divided into *k* subsets, and the holdout method is repeated *k* times. Each time, one of the *k* subsets is used as the test set and the other *k-1* subsets are put together to form a training set. Then the average error across all *k* trials is computed" [7]. Therefore, the "advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set *k-1* times " [7]. In contrast, the "disadvantage of this method is that the training algorithm has to be rerun from scratch *k* times, which means it takes *k* times as much computation to make an evaluation" [7]. For our program, we used a variation of k-fold cross validation. We "randomly divide[d] the data into a test and training set *k* different times" [7]. Then similar to the original k-fold cross validation, we averaged the results together after the *k* times of running the program. The advantage of using this variation is "that you can independently choose how large each test set is and how many trials you average over" [7].

Evaluating

Upon completion of testing the trained model, we want to be able to evaluate the model and see how accurate its predictions are. There are a couple of common techniques that can be used to do this: classification reports and confusion matrices.

A classification report "is used to measure the quality of predictions from a classification algorithm" [8]. The report provides information for three different classification metrics: precision, recall, and f1-score. These metrics "are calculated by using true and false positives, and true and false negatives" [8]. The positive and negative "are generic names for the predicted classes" [8]. Therefore, for our specific project, the positive and negative are represented by blue-green algae (bga), clear, and turbid. An example of a classification report is shown below. A true-negative is "when a case was negative and predicted negative" [8]. A true-positive is "when a case was positive and predicted positive" [8]. A false-negative is "when a case was positive but predicted negative" [8]. Finally, a false-positive is "when a case was negative but predicted positive" [8]. Precision is "the ability of a classifier not to label an instance positive that is actually negative" [8]. Therefore, it is defined for each class "as the ratio of true positives to the sum of true and false positives" [8]. Recall is "the ability of a classifier to find all positive instances" [8]. Therefore, it is defined for each class "as the ratio of true positives to the sum of true positives and false negatives" [8]. Lastly, the f1-score "is a weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is

0.0" [8]. This means that f1-scores "are lower than accuracy measures as they embed precision and recall into their computation" [8]. Therefore, the f1-score should be "used to compare classifier models, not global accuracy" [8].

```
Classification Report:
              precision    recall  f1-score   support

         bga       1.00      1.00      1.00         1
       clear       1.00      0.95      0.97        19
      turbid       0.88      1.00      0.93         7

    accuracy                           0.96        27
   macro avg       0.96      0.98      0.97        27
weighted avg       0.97      0.96      0.96        27
```

A confusion matrix "give[s] you a better idea of what your classification model is getting right and what types of errors it is making" [9]. For example, it "shows the ways in which your classification model is confused when it makes predictions" [9]. An example of a confusion matrix for our project is shown below. Each index represents a classification class. Index 0 represents "bga", index 1 represents "clear", and index 2 represents "turbid". For each slot in the matrix, the row represents what the image actually is, and the column represents how the image was classified by the model. Therefore, if the model classified the images with 100% accuracy, the matrix would only have numbers down its right-diagonal. Zeros would be filled in everywhere else. This means that every bga image was classified as bga, every clear image was classified as clear, and every turbid image was classified as turbid. In the example below, the model was almost 100% accurate. It only classified one image wrong. You can see that there was one image where the image was actually clear, but the model classified it as being turbid. Besides that, all of the bga and turbid images were classified correctly. Confusion matrices are very helpful because they give you insight into what specifically the model is getting wrong.

```
Confusion Matrix:
[[ 1  0  0]
 [ 0 18  1]
 [ 0  0  7]]
```

Labels are important in the evaluation process because in order to determine how accurate the model is, the program needs to know the original label for each image, as well as what the model actually predicted the image to be. If these two things match up, then the model is correct in its classification for the current image being evaluated. Without labels, the program would have no way of knowing if the model's predictions were accurate. Therefore, there would be no way to evaluate the model.

Methods and Procedures

Python is the language that is used to write the training and classifying programs for this project. This software was selected because it is supported by many open-source libraries for machine learning. These libraries allow us to not have to write machine learning algorithms

from scratch. Instead, we can access and use pre-written machine learning algorithms by downloading libraries and calling specific methods. Python is also very well known for its ability to support data science and data models. It is important that your chosen language can support these things when you are working with machine learning. This is because when you are doing machine learning, you are often dealing with large datasets. Many times you have to complete pre-processing on the data set in order to get it into a format that the machine learning model can import and read. Python contains many functionalities that make manipulating large data sets more efficient. These characteristics of Python made it possible to complete the tasks that we wanted for this project.

As stated above, Python contains many open-source libraries to use for machine learning. The library that we used for this project was Keras. Keras is a deep learning library that is written in Python. The main advantage to Keras is that it is a high-level API that is written on top of a lower-level machine learning platform, TensorFlow [10]. This allows us to use sophisticated machine learning algorithms and ideas without knowing the specific details. The API is very easy to follow and understand, making it friendly for users that are beginners with machine learning. In addition to supporting regular neural networks, Keras also has functionality for special types of neural networks, such as convolutional neural networks (CNNs). It was important that the library that we chose was able to support CNNs because CNNs are the type of networks that are primarily used to classify images due to how their architecture is structured. The data that we fed into our program consisted of images of the Finger Lakes. Therefore, we wanted to use a CNN for our machine learning model and needed a library that supported one. These reasons are why we decided to use Keras as the machine learning library for this project.

Program 1: *HABs_Train.py*

The first program that I wrote for this project was the code that was to train the neural network, *HABs_Train.py.* There were many parts that went into this program. However, a lot of these were completed by calling methods from either Keras' machine learning library or other libraries that were imported at the beginning of the program. Upon the completion of running this program, the machine learning model would be trained, tested, evaluated, and saved. The code for *HABs_Train.py* can be found in appendix A.

The first step was to load the dataset into the program. As stated above, the data for our project consisted of various images from the Finger Lakes. Therefore, we needed to load in each image from the folder where the dataset was saved. These would be the images that were used to train and test the model. This was done using Image's "open" method from the PIL library. This method takes the image's directory path as a parameter.

Next, labels needed to be created for each image. Labels let the model know which category each image belonged to. The labels were extracted from each image's file path as a String. The labels were then converted to numerical form using the LabelBinarizer from Scikit-learn's Preprocessing library. The LabelBinarizer's "fit_transform" method was called in order to do this conversion. This method takes the list that contains the string labels.

The third step was to split the data up into a training set and a test set. As stated above, the training set consists of the images that are used to actually train the CNN. The test set consists of a group of images that are set aside and are only used to test the CNN once it is finished training. In order to split the data up into these two sets we used the "train_test_split"

method from Scikit-learn's Model Selection library. In this method call, we were able to specify what percentage of the data we wanted to save for the test set. We decided to set aside 10% of the data to use for testing. Therefore, the training set consisted of 229 images, while the test set consisted of 26 images. We decided to use 10% of the data for the test set because the size of our data set is fairly small, so we wanted to be careful about setting aside too many pictures for testing, in order to provide the model with as much variation as possible. The "train_test_split" method randomly selected the images from the overall dataset for these groups and saved them into the specified variables. In order to run, this method needs the data array that contains the images, the array that contains the labels for the images, and the specified percentage for the size of the test set.

Next, it was time to define the architecture of the CNN. Keras has two kinds of models available: the Sequential model and the Model class used with the functional API [11]. The Sequential model is "the simplest type of model" [10]. It is a "linear stack of layers" [10]. The Model class that is used with the functional API is "for more complex architectures" and allows "arbitrary graphs of layers" to be built [10]. We decided to use the Sequential model for the CNN because it was the easiest to use. We first called the constructor of the Sequential class in order to create a new instance of the model. Once this was complete, we called the "add" method on the model. This method allowed us to add layers to the architecture of the model. With each call to "add", we are able to specify parameters for what goes in that specific layer. The Keras API goes into depth on the details of these parameters. The architecture structure that I decided on for my project was recommended by Adrian Rosebrock in a tutorial he posted on how to use machine learning in Python [4]. He said that this is a very basic CNN structure. In the future, I want to return to the architecture of our CNN and do experiments to find the optimal architecture for our specific problem: detecting HABs in the Finger Lakes. Currently, this architecture is able to classify the images at a reasonable accuracy. However, we have not tried to add or subtract different types of layers from the CNN's architecture in order to see how this would affect our model's overall performance level. By performing these experiments, we would know that we have the most optimal model possible for our project.

Once the architecture of the model was specified, it was time to compile the model. This meant that the model architecture had to be configured into a format that would allow it to be trained. This was done through Keras' "compile" method. In order to run, an optimizer and a loss function need to be specified. In addition, it also needs to be given a metric. We used the Adam optimizer, the categorical cross-entropy loss function, and the accuracy metric. All of these were used in the machine learning tutorial that was mentioned above [4]. Keras does offer other options, but we decided to use what was given in the tutorial. In the future, we can experiment with different optimizer and loss functions to see if these improve our results.

After the model was compiled, it was ready to be trained. The training was done through Keras' "fit" method. This method takes the images and labels of the training set, along with the images and labels from the test set (the validation data). In addition, the number of epochs and the batch size needs to be specified.

Upon completion of the model's training, it was time to use the model to predict the classifications of the images in the test set. In order to do this, we called Keras' "predict" method. This method takes images from the test set, along with the batch size.

When the model was done classifying the images that were set aside for the test set, it was time to evaluate the model and see how it did. We did this by using Keras' "evaluate"

method. This method needed the images and the labels from the test set. We were also able to receive the model's classification report and confusion matrix by calling the "classification_report" and "confusion_matrix" methods from Scikit-learn's Metrics library. Both of these methods needed model's predictions of the test images and the labels from the test set. The classification report also needed to be given the names of the three classification categories: bga, clear, and turbid.

Lastly, we saved our trained model and its architecture by calling Keras' "save" method. In the method call, we needed to specify the name that we wanted to call our model. This method saved the model in a single HDF5 file, and gave us the ability to re-create our model in a program that was separate from the original program where the model got trained. We could load the model into a program by calling Keras' "load_model" function in the particular program. This became useful for us when we wanted to write a separate program, *HABs_Classify.py*, that used this trained CNN model to classify new images of the Finger Lakes.

Program 2: *HABs_Classify.py*

The second program that I wrote for this project, *HABs_Classify.py*, consists of the code that classifies and sorts new images of the Finger Lakes. It does this by using the CNN that was constructed and trained in our first program, *HABs_Train.py.* It also generates a report on the classifications, which is saved in a text file. The code *HABs_Classify.py* can be found in appendix B.

The first thing that we did for this program was load in the model that we wanted to use in order to classify the images. We did this by using Keras' "load_model" method. We entered the name of the model that we wanted as a parameter.

Next, we had to prepare the input for the program. First, we had to load in the set of images that were to be classified by the model. Similar to the process that took place in the *HABs_Train.py* program, we needed to load in each image from the folder where the data was saved. This was done by calling the "load_img" method from Keras' Preprocessing Image library. We needed to specify the image's directory path in order to call this method. After this, we had to complete a few pre-processing steps in order to make sure that each image was in a format that the Keras machine learning methods could read.

Once all of the images were in the correct configuration, it was time to predict their classifications using the loaded CNN model. This was done by calling Keras' "predict" method. We entered the data set of images that we wanted to classify as a parameter.

Following the completion of the predictions, the program sorted each image and placed it in a folder depending on this classification. Before the program completed, it produced a report that summarized how each image was classified by the model. This report was saved as a text file and could be imported into Excel for easy access.

Results

The CNN model trained in *HABs_Train.py* has an average accuracy of 89.499%. This accuracy was determined through the k-fold cross validation variation technique that was discussed in the background section. We used a *k* value of 100. Therefore, we ran the model 100 separate times. Each time, 10% of the data set was randomly set aside to test the network after it was trained on the remaining 90%. The accuracy of the test run was recorded after each

trial run. Upon the completion of the 100 trials, we calculated the average accuracy score, which came out to be 89.499%. The results of the 100 trials can be found in appendix C.

With the model recording nearly a 90% accuracy rate, we can say that this model is of reasonable quality. As explained in the above sections, the accuracy of the various trials varied because each trial was working with a different combination of images for the training and test sets. This caused a fluctuation in the results. However, overall, the model was able to classify the images correctly with on average 2 or 3 mis-classifications out of a test set that had 26 images. Throughout the 100 trials, there were 8 instances where the test set was classified with 100% accuracy. This illustrates that it is possible for all of the images in a set to be classified correctly. Although it is rare.

There were a few patterns found in images which the model had trouble classifying. One was that the CNN had a difficult time telling the difference between turbid and clear images. This combination was the one that the model had the most issues with. For example, the lowest accuracy that the model recorded out of the 100 trials was 65.28%. In this trial, the model mis-classified 7 out of the 11 turbid images that were in this test set. It thought that these 7 images were clear. After analyzing the dataset, it is understandable to see why the model has trouble determining the difference between some turbid and clear images. This is because some of the images share a few common characteristics that the model could be using for its categorization. For example, the clear images contain a bunch of rocks that are at the bottom of the lake. In some turbid images, it is possible to see some of the rocks at the bottom of the lake through the turbidity. This can be seen in the images below. The image on the left is an example of a clear image, while the image on the right is an example of a turbid image. In the turbid

image you can see some rocks. Therefore, the rocks in the turbid image could have been one of the reasons why it was mis-classified. It could be possible that the model associates the presence of rocks in the image with images that are classified as clear. It is hard to say for certain because the model is a “black box” and it is difficult to try to understand how it is making its decisions. However, this could be an explanation for why the model often mis-labeled turbid and clear images.

The bga images were often classified correctly. This was encouraging because bga is ultimately what we want our program to identify and sort out. However, when a bga did get mis-classified, it was often with a clear image. From my observations, I think the reason for this would be similar to the reason why the turbid images were getting confused with the clear images: the rocks. There are some bga images where the algae is so light, that the rocks are

visible in the image. The light bga appearance, would make it more difficult for the model to identify the algae amongst all of the rocks. Therefore, it is possible that it would not be a big enough factor for it to classify it as bga over clear. An example of this situation is shown below. A clear image is shown on the left, a light bga image is shown in the middle, and a heavy bga image is shown on the right.

After I received 89.499% accuracy for the model after running *HABs_Train.py* 100 times, I ran *HABs_Classify.py* with 10 new images that the model did not see during any of the trial runs. This new image set consisted of 4 clear images, 3 turbid images, and 3 bga images. Although the set was very small, it consisted of a variety of types of images. For example, it consisted of images with heavy bga and light bga. It had turbid images that showed rocks and that did not show rocks. Running the model from *HABs_Classify.py* on this data set would help give us insight into how the model was sorting things. When I ran *HABs_Classify.py*, 9 out of the 10 images were sorted correctly. Therefore this 90% accuracy was consistent with the model's average accuracy that was calculated after the 100 trial runs with *HABs_Train.py.* The image that got mis-classified was a turbid image, "turbidTest3.jpg". It was classified as a clear image. This image is shown below. In this image, some rocks can be seen through the turbidity. Therefore, the reason why this image was mis-classified could be similar to the reasons stated above.

The *HABs_Classify.py* program creates a report upon its completion that supplies information about how the program classified each image. The report sorts the images from the least confident to the most confident in the program's prediction of the images. The last three columns show us the percentage (out of 1) of how confident the program was at classifying the individual image to the three possible categories of bga, clear, or turbid. This information allows us to see how close the program's decision was in predicting an image's classification. The report that was created for the 10 new images is below. The results of the classifications of the 10 new images are consistent with the report. This is because since the

image, “turbidTest3.jpg” is the first image listed, it means that this was the image that the model was least confident in classifying. The model was 64% confident that “turbidTest3.jpg” should be classified as clear, while it was 35% confident that it should be classified as turbid. The second least confident image was “bgaTest2.jpg”. This makes sense because this image consists of light bga. Even

MaxValueSorter	ImageName	Classification	bgaConfidence	clearConfidence	turbidConfidence
6412	turbidTest3.jpg	clear	0.0042015	0.6412936	0.3545049
6613	bgaTest2.jpg	bga	0.66132885	0.18001644	0.1586547
7518	turbidTest2.jpg	turbid	0.06914653	0.17897767	0.7518758
7583	clearTest2.jpg	clear	0.008727393	0.75831276	0.23295991
9139	bgaTest1.jpg	bga	0.91399425	0.061624154	0.024381688
9385	clearTest3.jpg	clear	0.003684779	0.93855864	0.057756506
9463	turbidTest1.jpg	turbid	0.001949151	0.051695302	0.9463555
9668	clearTest4.jpg	clear	0.00019394	0.9668657	0.03294031
9804	clearTest1.jpg	clear	0.017127268	0.9804048	0.002467942
9815	bgaTest3.jpg	bga	0.981588	0.012204857	0.006207165

though the model was still able to classify it correctly, it was not as confident as the other images. All 10 of the images that were classified by *HABs_Classify.py,* and are detailed in the report above, can be found in appendix D.

Overall, the model produces satisfactory classification of images of the Finger Lakes. The tests show that there are still some types of images that the model has trouble classifying correctly every time. However, it is accurate enough where it can be used to initially sort the data. This means that the model and program can do the initial sorting, but that the user should still go through with the generated report and spot check the images that the program was least confident about, in order to make sure that they were placed in the correct folder.

Next Steps

There are a few steps I would take next if I were to continue with this project in the future. These steps include getting the program to classify images that are not as uniform as the current images, continue to limit the biases that could take place when training the network, create a more diverse and extensive data set to train the model, complete more experiments with the architecture of the model, and fixing a minor logistical error that is currently in the program.

One step I would take to enhance this project would be to train the program so that it can recognize HABs in images that are not as uniform as the current images that the program is working with. Currently, the images that the program is trained to classify are images that are all taken by a camera that is placed at the same location and height. Therefore, for these images, the program is always looking at the same area of water. The space is consistent and only contains water. The model does not have to worry about blocking objects out of the background, such as the shoreline, docks, or trees. However, it would be helpful if the program was able to classify images that include bigger areas and are taken from varying heights. This is because a common technique that researchers use to collect data of HABs is to take high quality images and videos of the Finger Lakes using drones. When taking pictures with a drone, it is hard to get the drone to be at the same exact altitude and location every time. In addition, the drone images contain more background because they are taken at a higher altitude than the camera images that are currently being used. If our program was able to take in a variety of images from different altitudes and locations with the possibility of containing objects other than just water, then it allows scientists and researchers more flexibility with the type of data that they can use.

Another step I would take to improve this project would be to continue to look into how to limit the biases that could take place when training the network. As stated in the results section, when a machine learning model is trained, it is possible for it to create biases towards specific characteristics in the data that does not have anything to do with whether the image contains HABs or not. Therefore, we want to make sure that the images are being classified on its characteristics of containing HABs. We do not want the model to make its predictions based on other characteristics that the images could have in common, such as shadow placement, sun glares, and the time stamps that are at the bottom of the images. One step I would take to improve the biases would be to complete some extra pre-processing steps on the images before they are imported into the model. This would involve removing the time stamps from the images. The time stamps could have skewed the classification results if a lot of the images contained the same or a similar time stamp. Another pre-processing technique would be to remove the glares from the images. Similar to the time stamps, these could have skewed the results as well.

In addition to completing additional pre-processing steps, another way to limit the biases would be to train the model with a more varied and extensive data set. In order to do this, a larger data set with more variety would need to be established. Currently, the model is trained on a data set that is fairly small and not very diverse. The present data set was compiled using a couple months of data. In addition, a lot of the images were taken at the same times during the day, so there is not a lot of variation with the shadows that are in the images. Within this data set, we only have a few good examples of HABs. And these were all taken on the same day. HABs are what this program is trying to classify, so it would be a good idea to continue to add new images of HABs to the dataset. Overall, it would make the classification function more accurate and able to recognize a broader range of HABs. The data set can be expanded by continuing to collect images of data and HABs and adding them to the existing data set. Then this new data set could be imported into *HABs_Train.py* in order to create and save a new model that is trained using this new data set.

A fourth step I would complete if I were to continue this project would be to conduct more experiments with the architecture of the machine learning model. As stated in the

methods and procedures section, the architecture structure that I used for this project was provided in a tutorial by Adrian Rosebrock [4]. With more time, I would like to go back and determine if there are other architecture combinations that have the ability to predict the classifications more accurately, once it is trained using the data set. This would involve either adding or removing layers from the model. By completing experiments to find the optimum architecture combination for the model, we are making sure that we are using the most accurate model as possible in order to classify the images. The current model is relatively accurate, but it would be ideal if we could be certain that we were using the best model.

Lastly, there is one main logistical problem that I would fix. The complication is that whenever the user imports images into the program, *HABs_Classify.py*, they must make sure that the main image folder contains three subfolders, and that the images are in at least one, if not all of these folders. The reason for why three subfolders are needed is due to how the structure of the program is designed. I built my program off of a couple of machine learning tutorials that I discovered. One of the characteristics of these tutorials is that the base picture folder always contains three subfolders. This is because this tutorial was loading in the images from three sorted folders. This organization allowed the program to easily use the images for training, testing, and evaluating the model. I have not figured out how to make it so the picture folder does not need subfolders. I think it has to do with one of the libraries that the tutorial used. In this situation, I did not use the most convenient and flexible solution for the problem, but the program is still able to work if the correct precautions are taken. However, they are not convenient for the user, and if the user does not carefully follow the directions, the program will give an error. Given more time, I would research ways on how to fix these inconveniences for the user.

What I learned

While creating this machine learning program that classifies HABs in the Finger Lakes, I learned many concepts and practices that will help me with future projects. I had to learn the language Python and become familiar with its syntax. This was my first time using Python for a program this large, and I was not very comfortable with its structure. In addition to teaching myself a new language, I also had to use a variety of Python libraries in my program. These libraries allowed me to use methods that were written by other programmers. However, understanding what each specific library did and how to use it correctly was often challenging, and gave me the opportunity to strengthen my problem solving skills. Lastly, I increased my knowledge in machine learning. There are many components that go into successfully completing a machine learning project. First you have to understand the idea behind machine learning and its history. Then, it is helpful to be able to know the science behind the structure and architecture of the machine learning models. Next, it is important to learn the process of how to pre-process and load your data, train the model, test it, and evaluate it.

The biggest lesson that I learned throughout the process of creating this program is that when you are completing a project that involves machine learning, machine learning only makes up a fraction of the project. The majority of the time will be spent working on other logistics of the program, such as pre-processing and formatting your data so that it can be read by the machine learning model, or coding the program to do what you want with your data once it is all classified. As stated above, the machine learning aspect of the process includes the configuring of the model's architecture, and training, testing, and evaluating the model. Once

the predictions are made using the trained model, the machine learning aspect of the project is complete. Next, it is time to take the predictions and use them to make an output that is useful for the user. For our project, this consisted of figuring out how to copy the images into two sorted folders or directories. This took a lot more steps and problem solving than I anticipated. Some of the things that I had to learn in order to do this were: determine how to save a trained model and load it into a separate program, determine how to make the program work for both Mac and Windows users, determine how to convert the image into the correct format that the machine learning model would take, and determine how to convey the predicted information for each image in a report. All of these tasks took a significant amount of time to solve. Each one had their challenges and led me to either learn a new technique in Python, or become familiar with a new library.

Conclusion

Currently, there are many researchers and scientists that are studying HABs in order to better understand their behaviors and patterns. The methods that they presently use are very tedious and time consuming. The purpose of this project was to implement one of the applications that would help their work become more efficient. Our goal was to develop a program that takes in a set of images that were taken of the Finger Lakes and sorts them into separate folders based on the images' characteristics.

The result of our project lines up with the initial objectives. We were able to create a program that moves images into groups based on characteristics that they showed of containing bga, being clear, or being turbid. The accuracy of the program was sufficient with an average of 89.499%. The program also generates a report that allows you to look at information about how the program classified the images. In order to get the most accurate results, it is recommended to use the program to do the initial sorting of the image set that is imported into the program. After the program is complete, open up the newly produced report and find the images that the program was least confident in classifying. Then go to the sorted folders and quickly spot check these images to make sure that they were placed in the correct folder. Even though this new tool still involves some manual work at the end, it is still much more efficient than having to sort every individual image by hand. Overall, it is anticipated that researchers will only have to fix the placement of a few images. The number of images they will have to move at the end depends on how large their image set is.

The software features of this program are very basic. The program runs by using the command line. Before the program starts, the user needs to set up the proper directory structure. In the future, it would be ideal to design a graphical user interface for this program. This would make it much more friendly for the user. However, the way that the program is set up right now, it still satisfies the needs of the user, and allows researchers to do their job more efficiently. The suggestions I made in Next Steps would significantly improve the overall process making it more convenient and accurate for all researchers studying Harmful Algal Blooms.

Appendix A

HABs_Train.py

```
# Author: Bizzy Moore
# Hobart and William Smith Colleges
# CPSC 450 - Independent Study
# Spring 2020

# ----------------- Program Notes -----------------------------
# Trains and saves a Convolutional Neural Network (CNN)
# using the Keras library.
# The model is trained using images from the Finger Lakes.

```

```
# --------------- Import Libraries --------------------------
# import the necessary libraries and packages
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.optimizers import Adam
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from PIL import Image
from imutils import paths
import numpy as np
import argparse
import os
from keras.models import load_model
import shutil
import platform
from pathlib import PurePosixPath
from pathlib import PureWindowsPath
from operator import itemgetter

```

```
# ----------------- Get Command Line Arguments ----------------------
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", type=str, default="ImportedPics",
    help="path to directory containing image dataset that is to be imported")
ap.add_argument("-m", "--model", type=str, default="HABs_CNN_Model_FINAL.h5",
    help="name of CNN model that will be saved")
args = vars(ap.parse_args())

```

```
# ----------------- Load Dataset -------------------------------------
print("\n[INFO] loading images...")

# grab all image paths in the input dataset directory
# initialize lists of images and corresponding class labels
imagePaths = paths.list_images(args["dataset"])
data = []
labels = []

# loop over our input images
for imagePath in imagePaths:
    # load the input image from disk
    # resize it to 32x32 pixels
    # scale the pixel intensities to the range [0,1]
    # divide by 255 because 255 is the max rgb value for each pixel, 0-255
    # updatae the image list
    image = Image.open(imagePath)
    image = np.array(image.resize((32, 32))) / 255.0
    data.append(image)

    # extract the class label from the file path
    # update the labels list
    label = imagePath.split(os.path.sep)[-2]
    labels.append(label)

# encode the labels, converting them from strings to integers
lb = LabelBinarizer()
labels = lb.fit_transform(labels)

# -------------------- Split Data -------------------------------------
# split data set into a training set a test set
(trainX, testX, trainY, testY) = train_test_split(np.array(data),
    np.array(labels), test_size=0.10)

# --------------------- Define Model ------------------------------------
# define our Convolutional Neural Network architecture
model = Sequential()
model.add(Conv2D(8, (3, 3), padding="same", input_shape=(32, 32, 3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(16, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(32, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(3))
model.add(Activation("softmax"))

```

```
    # ------------------ Compile Model ----------------------------------
    # train the model using the Adam optimizer
    opt = Adam(lr=1e-3, decay=1e-3 / 50)

    # compile model
    model.compile(loss="categorical_crossentropy", optimizer=opt,
        metrics=["accuracy"])

    # -------------------  Run Trial ----------------------------------
    print("\n[INFO] training network...")

    # fit/train model
    trial = model.fit(trainX, trainY, validation_data=(testX, testY),
        epochs=50, batch_size=32)

    # get model's prediction
    predictions = model.predict(testX,batch_size=32)

    print("\n[INFO] results:")

    # get evaluation of model
    evaluation = model.evaluate(testX, testY, verbose=0)
    print("%s = %.2f%%" % (model.metrics_names[1], evaluation[1]*100))

    # get classification report
    classReport = classification_report(testY.argmax(axis=1),
        predictions.argmax(axis=1), target_names=lb.classes_)
    print("\nClassification Report Trial: ")
    print(classReport)

    # get confusion matrix
    print("Confusion Matrix Trial: ")
    print(confusion_matrix(testY.argmax(axis=1), predictions.argmax(axis=1)))

    # ---------------------- Save Model ----------------------------------
    # save trained model and architecture to single file
    model.save(args["model"])
    print("\nSaved model to disk.\n")
```

Appendix B

HABs_Classify.py

```
# Author: Bizzy Moore
# Hobart and William Smith Colleges
# CPSC 450 - Independent Study
# Spring 2020

# ----------------- Program Notes -----------------------------
# Classifies images of the Finger Lakes into three categories:
# blue-green algae (BGA), clear, or turbid.
# Sorts the images into a folder according to the specific
# classification that they received.
# Produces a report that supplies information about how the
# model classified each image.

# --------------- Import Libraries --------------------------
# import the necessary libraries and packages
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.optimizers import Adam
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from PIL import Image
from imutils import paths
import numpy as np
import argparse
import os
from keras.models import load_model
import shutil
import platform
from pathlib import PurePosixPath
from pathlib import PureWindowsPath
from operator import itemgetter
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

```

```
41  # --------------- Determine Operating System --------------------
42  # get computer's operating system
43  opSys = platform.system()
44
45  # get seperator based on OS
46  # set OS flags
47  if (opSys == 'Linux'): # OS is Linux
48      seperator = '/'
49      isLinux = True
50      isMac = False
51      isWindows = False
52
53  if (opSys == 'Darwin'): # OS is Mac
54      seperator = '/'
55      isLinux = False
56      isMac = True
57      isWindows = False
58
59  if (opSys == 'Windows'): # OS is windows
60      seperator = '\\'
61      isLinux = False
62      isMac = False
63      isWindows = True
64
```

```
65  # --------------- Get Command Line Arguments  -----------------------------
66  # construct the argument parser and parse the arguments
67  ap = argparse.ArgumentParser()
68  ap.add_argument("-d", "--dataset", type=str, default="ImportedPics",
69      help="path to directory containing image dataset that is to be imported")
70  ap.add_argument("-c", "--classpics", type=str, default="ClassifiedPics",
71      help="name of the folder that holds the classified pictures")
72  ap.add_argument("-t", "--txtfilename", type=str, default="HABsClassReport",
73      help="name of the text file that will be created")
74  ap.add_argument("-m", "--model", type=str, default="HABs_CNN_Model_FINAL.h5",
75      help="name of CNN model that is loaded")
76  args = vars(ap.parse_args())
77
78  # --------------- Load Model ------------------------------
79  print("\n[INFO] loading model...\n")
80
81  # load pre-trained model into program
82  model = load_model(args["model"])
83
```

```
84  # ------------------ Load Dataset -----------------------
85  print("\n[INFO] loading images...\n")
86
87  # grab all image paths in the input dataset directory
88  imagePaths = paths.list_images(args["dataset"])
89
90  # initialize lists of source directories and images
91  imageSRCs = []
92  data = []
93
```

```
94  # loop over our input images
95  for imagePath in imagePaths:
96
97      # add image's path to source directory list
98      imageSRCs.append(imagePath)
99
100     # load the input image from disk
101     image = load_img(imagePath)
102
103     # reside the image to 32x32 pixels
104     imageResize = image.resize((32,32))
105
106     # convert image to a numpy array
107     imageToArray = img_to_array(imageResize)
108
109     # scale image to the range [0,1]
110     # divide by 255 because 255 is the max rgb value for each pixel, 0-255
111     imageDivide = imageToArray / 255.0
112
113     # expand the dimensions of image
114     imageExpand = np.expand_dims(imageDivide, axis = 0)
115
116     # add the image to the data list
117     data.append(imageExpand)
118
119 # put the complete data list in a vstack
120 dataVStack = np.vstack(data)
121
```

```
122 # ---------- Classify the Images --------------
123 print("[INFO] classifying images...\n")
124
125 # generate model's classification predictions for each input image
126 # outputs array for each image with the % of confidence for each classification label
127 predictions = model.predict(dataVStack)
128
129 # gives us the model's classification for each input image
130 # takes the index that contains the max % value from each image's predictions array
131 classifications = predictions.argmax(axis=1)
132
```

```
133  # ----------- Save Calssified Images in Sorted Folders -----------
134  print("[INFO] sorting images...\n")
135
136  # create the main list that will hold all of the images' classification information
137  totalList = []
138
139  # copy each image into the correct folder based on their new classification
140  for imageSRC, predictedNums, classNum in zip(imageSRCs, predictions, classifications):
141      # get filename of image
142      if (isWindows == False):
143          # use PurePosixPath for Mac and Linux os
144          filenameImg = PurePosixPath(imageSRC).name
145      if (isWindows == True):
146          # use PureWindowsPath
147          filenameImg = PureWindowsPath(imageSRC).name
148
149      # get max % value from images predicted array
150      maxVal = max(predictedNums)
151
152      # convert value to a string
153      strMaxVal = str(maxVal)
154
155      # get the first 4 decimal places from the max % value
156      # will be used to help sort the images based on the model's confidence level
157      maxSorter = strMaxVal[2:6]
158
```

```
159        # get the classification folder and name for image
160        if(classNum == 0):
161            # change path to bga folder
162            classifiedFolder = "bgaClassified"
163            className = 'bga'
164        elif(classNum == 1):
165            # change path to clear folder
166            classifiedFolder = "clearClassified"
167            className = 'clear'
168        elif(classNum == 2):
169            # change path to turbid folder
170            classifiedFolder = "turbidClassified"
171            className = 'turbid'
172        else:
173            # set to miscellaneous
174            classifiedFolder = "miscellaneous"
175            className = 'miscell'
176
177        # create list with classification information for image
178        listCluster = [maxSorter,filenameImg,className]
179
```

```
        # add the predicted numbers for each category to the list
        for num in predictedNums:
            listCluster.append(num)

        # add images list cluster to the main list
        totalList.append(listCluster)

        # create a new filename that the image will be saved as in its classified folder
        # this new filename contains the maxSorter
        # this allows the images to be sorted by confidence level in its classified folder
        newFilename = maxSorter+'-'+filenameImg

        # get the name of the folder that holds all of the classified images
        classifiedpicsfolder = args["classpics"]


        # create the new file destination for the image
        # make sure that the correct current directory (cd) is set
        # goes to relative path of a file in a subdirectory of the cd
        dstFile = classifiedpicsfolder+seperator+classifiedFolder+seperator+newFilename

        # copy file into correct classification folder
        shutil.copy(imageSRC,dstFile,follow_symlinks=True)

```

```
# ------------------- Generate Report -------------------------------
print("[INFO] generating report...\n")

# sort the list clusters that are in totalList in ascending order
totalList = sorted(totalList, key=itemgetter(0))

# create name for report
textfilename = args["txtfilename"] + ".txt"

# write to report file
file = open(textfilename,"w")
file.write(",MaxValueSorter,ImageName,Classification,bgaConfidence,clearConfidence,turbidConfidence\n")

# write each list cluster from total list into report
for cluster in totalList:
    clusterString = ""
    for thing in cluster:
        clusterString = clusterString + "," + str(thing)
    clusterString = clusterString + "\n"
    file.write(clusterString)
file.close()

print("Program complete. \n")
```

Appendix C

Results of the 100 Trials:

- The first column states the specific trial number.
- The second column displays the model's accuracy for classifying the images in the test set for that specific trial number.
- The last three columns display information about how the test images for each separate category were classified by the model.
    - The number before the first \ represents the amount of images that were correctly classified as that category.
    - The number(s) after the first \ represent the number of images of that specific category that were classified incorrectly.
    - The letters after the numbers represent the particular category that the images were mis-classified to.
        - BGA -> "b"
        - Clear -> "c"
        - Turbid -> "t"
    - Example:
      For trial #1, the model classified 84.62% of the test set correctly.
      The test set consisted of a total of 26 images: 1 BGA, 16 clear, and 9 turbid.
      The model classified its one BGA image correctly.
      It did not mis-classify any BGA images.
      The model classified 13 of the clear images correctly.
      It mis-classified 3 clear images as turbid.
      The model classified 8 of the turbid images correctly.
      It mis-classified 1 turbid image as clear.

Trial Number	Accuracy %	BGA	Clear	Turbid
1	84.62	1\0	13\3t	8\1c
2	88.46	1\1c	18\0	4\2c
3	96.15	2\0	14\1b	9\0
4	92.31	1\1c	15\0	8\1c
5	88.46	2\0	16\0	5\3c
6	96.15	1\0	21\0	3\1c
7	96.15	2\0	14\0	9\1c
8	65.28	1\1c	12\1t	4\7c
9	88.46	2\2c	11\1	10\0

10	100	2\0	14\0	10\0
11	96.15	3\0	14\0	8\1c
12	92.31	3\0	13\2t	8\0
13	92.31	3\1c	15\0	6\1c
14	80.77	1\1c	14\2t	6\2c
15	96.15	2\0	16\0	7\1c
16	88.46	2\0	15\2t	6\1c
17	96.15	5\0	15\0	5\1c
18	96.15	2\0	17\1b	6\0
19	80.77	1\1t	10\1t	10\3c
20	92.31	2\0	17\0	5\2c
21	88.46	3\0	13\3t	7\0
22	96.15	1\0	19\0	5\1c
23	92.31	2\0	18\0	4\2c
24	92.31	1\0	15\1b	8\1c
25	96.15	3\0	17\0	5\1c
26	80.77	1\0	14\0	6\5c
27	80.77	1\0	15\1b	5\4c
28	92.31	1\0	18\0	5\2c
29	88.46	1\0	19\0	3\3c
30	88.46	1\0	10\1t	12\2c
31	100	2\0	17\0	7\0
32	88.46	4\2c	18\0	1\1
33	96.15	2\0	20\0	3\1c
34	88.46	1\0	9\2t	13\1c
35	84.62	2\0	13\2b\2t	7\0
36	100	5\0	10\0	11\0
37	84.62	2\0	10\3t	10\1c
38	84.62	1\1c	16\2t	5\1c
39	88.46	1\0	16\2t	6\1c
40	73.08	1\0	13\7t	5\0
41	88.46	4\0	11\2t	8\1c
42	92.31	1\0	15\0	8\2c

43	84.62	2\0	18\1t	3\2c
44	88.46	2\0	11\0	10\3c
45	92.31	2\0	18\0	4\2c
46	88.46	2\0	13\0	8\3c
47	96.15	2\0	17\0	6\1c
48	84.62	2\3c	13\0	7\1c
49	100	1\0	19\0	6\0
50	92.31	2\1c	19\0	3\1c
51	96.15	2\0	16\1t	7\0
52	100	1\0	19\0	6\0
53	88.46	1\0	15\0	7\3c
54	76.92	2\0	15\0	3\6c
55	96.15	1\0	14\0	10\1c
56	88.46	2\0	18\1t	3\1c
57	80.77	3\0	16\0	2\5c
58	96.15	4\0	17\0	4\1c
59	80.77	1\1c\1t	13\1t	7\2c
60	92.31	5\0	14\0	5\2c
61	88.46	1\1c	17\0	5\2c
62	84.62	2\1c	17\2t	3\1c
63	84.62	1\1c\1t	15\0	6\2c
64	88.46	1\1t	16\2t	6\0
65	92.31	1\0	13\1t	10\1c
66	88.46	2\0	10\1t	11\2c
67	80.77	1\0	14\0	6\5c
68	88.46	1\0	15\3t	7\0
69	92.31	2\0	15\0	7\2c
70	92.31	2\0	13\1t	9\1c
71	88.46	1\0	19\1t	3\2c
72	92.31	1\1c	17\1t	6\0
73	84.62	1\1c	15\1b	6\2c
74	88.46	4\1c	13\1t	6\1c
75	88.46	1\1t	15\0	7\2c

76	88.46	1\1c	14\1t	8\1c
77	80.77	3\0	13\1t	5\4c
78	84.62	2\0	12\3t	8\1c
79	100	1\0	13\0	12\0
80	88.46	1\1c	13\2t	9\0
81	76.92	2\1c\1t	8\4t	10\0
82	88.46	1\1t	15\0	7\2c
83	92.31	3\0	14\0	7\2c
84	92.31	4\0	12\0	8\2c
85	76.92	1\0	14\1t	5\5c
86	96.15	4\0	12\1b	9\0
87	100	2\0	11\0	13\0
88	96.15	1\0	12\0	12\1c
89	96.15	3\0	16\1t	6\0
90	92.31	3\1t	14\0	7\1c
91	84.62	1\0	12\2t	9\2c
92	100	3\0	14\0	9\0
93	92.31	2\0	16\0	6\2c
94	84.62	1\0	18\1b	3\3c
95	73.08	3\1c	15\0	1\6c
96	92.31	3\0	16\0	5\2c
97	96.15	2\0	13\1t	10\0
98	84.62	3\1c	9\1b\1t	10\1c
99	84.62	2\0	17\1t	3\3c
100	88.46	1\0	18\0	4\3c
Average Accuracy:	**89.499%**			

Appendix D

10 Images Classified by *HABs_Classify.py*

clearTest1.jpg

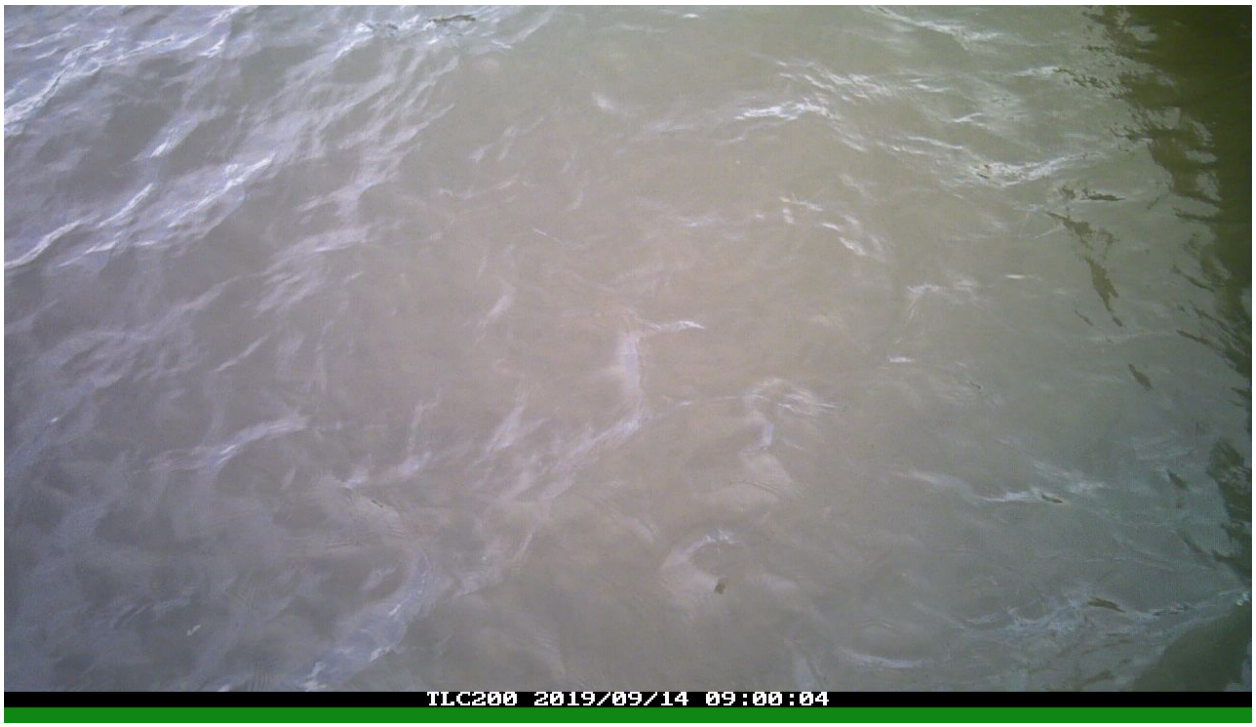
clearTest2.jpg

clearTest3.jpg

clearTest4.jpg

turbidTest1.jpg

turbidTest2.jpg



turbidTest3.jpg

bgaTest1.jpg

bgaTest2.jpg

bgaTest3.jpg

Works Cited

[1] https://www.noaa.gov/what-is-harmful-algal-bloom

[2] Bridgeman, Stina.” 08-machinelearning.pdf.” PDF file.

[3] https://pathmind.com/wiki/neural-network

[4] https://www.pyimagesearch.com/2019/01/14/machine-learning-in-python/

[5] https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

[6] https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss

[7] https://www.cs.cmu.edu/~schneide/tut5/node42.html

[8] https://muthu.co/understanding-the-classification-report-in-sklearn/

[9] https://machinelearningmastery.com/confusion-matrix-machine-learning/

[10] https://keras.io/about/

[11] https://keras.io/api/models/