Detecting Damaged Buildings based on Post- Hurricane Satellite Imagery

Jake Wojcik

1 Problem & Context

After a hurricane, damage assessment is essential to emergency workers and first responders so resources can be planned and designated accordingly. One way to gauge the damage extent is to detect and quantify the number of damaged buildings, which is typically done by ground survey methods. This process can be labor intensive and time-consuming. With satellite imagery readily available today, the goal of this study is to improve the efficiency and accuracy of building damage detection with image classification algorithms.

2 Target Clients

The primary clients of this project targets emergency workers, first responders, and politicians in charge of allocating disaster relief resources. Not only will this project, if successful, save time and money for disaster workers, It will be beneficial for those affected by the natural disaster.

3 Data

The data are satellite images from Texas after Hurricane Harvey divided into two groups (damage and no_damage). The dataset is structured in the following format:

- **train_another:** the training data; 5000 images of each class
- **validation_another:** the validation data; 1000 images of each class
- **test_another:** the unbalanced test data; 8000/1000 images of damaged/undamaged classes

- **test:** the balanced test data; 1000 images of each class

All images are in JPEG format. The dataset is available for download via the following link: https://www.kaggle.com/kmader/satellite-images-of-hurricane-damage

4 Data Wrangling

Because the data came from a Kaggle dataset there wasn't much cleaning or transforming of the images to be done. I used the pathlib library in order to extract relevant features of the images into a dataframe. Using the pathlib library, I pulled the images into separate dataframes by train, validation, and test. The resulting dataframes included the following columns: file path, damage, data_split, location, longitude, latitude, and image in matrix format. With this information for each image included in a dataframe it made it much easier to explore the data.

First, it's important to check for duplicate images in the dataframes. Duplicated images could lead to data leakage within the model, ultimately resulting in worse model accuracy and possibly overfitting problems. Out of the 12,000 train and validation images, I found 9476 unique locations in which the images were taken. I decided to investigate further to ensure there were no duplicate images. It turns out there were several images taken at the same location but one before the hurricane (undamaged) and one after the hurricane (damaged). Figure 1 shows the same location, one damaged and one undamaged. It's obvious the damaged image has evidence of flooding and building damage compared to the non-damaged image.

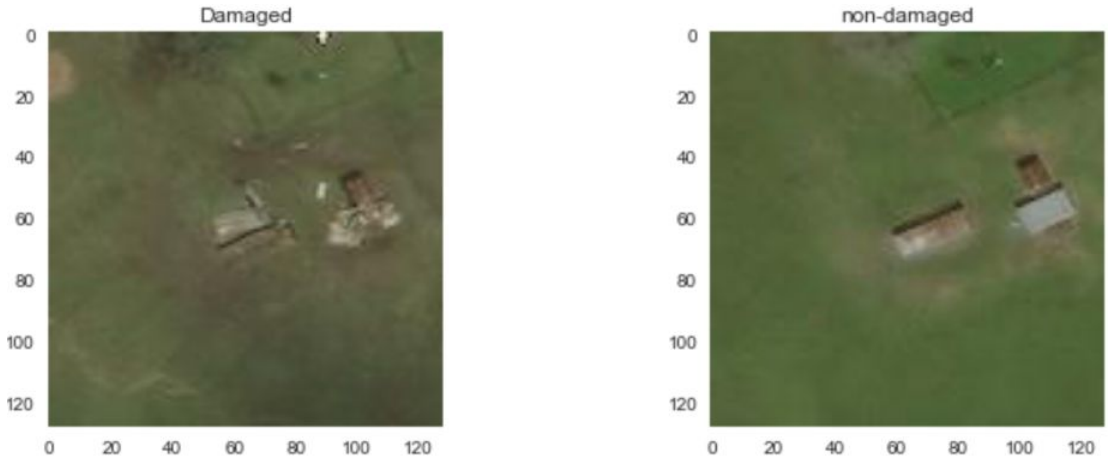

Figure 1: Comparison of the same location before (right) and after (left) Hurricane Harvey.

5 Exploratory Data Analysis

First, it's valuable to get a good idea of the size of the images in the and the distribution between classes. Again, because the dataset already had the initial preprocessing steps completed, every image was 128 x 128 X 3 giving an aspect ratio of 1. Our train set was evenly split between classes with 5,000 damaged images and 5,000 undamaged images.

5.1 Location

Next, it's important to gain a better understanding of how these images were distributed spatially. In order to achieve that, Plotly was used to plot each image overlaid on a map of the greater Houston area labelled by damaged and undamaged (Figure 2). There are around 4 different spatial clusters in this dataset. Interestingly, the Beaumont area only has damaged buildings. Also, if you were able to zoom in on the map, you'd find most of the damaged buildings occur on the banks of rivers and waterways.

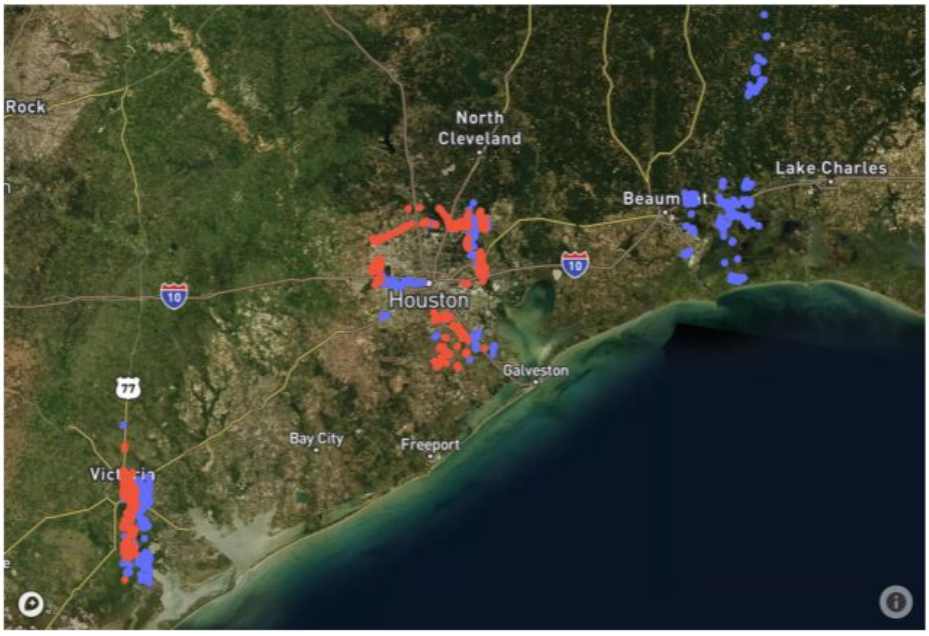

Figure 2: Map of the greater Houston area with labeled images. Red represents undamaged images and blue represents damaged images.

5.2 Color Distribution

Color space is represented by three different channels Red, Green, and Blue. The three primary colors are added to produce 16,777,216 distinct colors in an 8-bit per channel RGB system. Color is usually thought of as an important property of objects and is often used for segmentation and classification in computer vision. Color could be used as a feature vector in a neural network in order to distinguish between classes. We found the undamaged images have more color intensity in all three channels by taking the mean value in each channel (Figure 3).

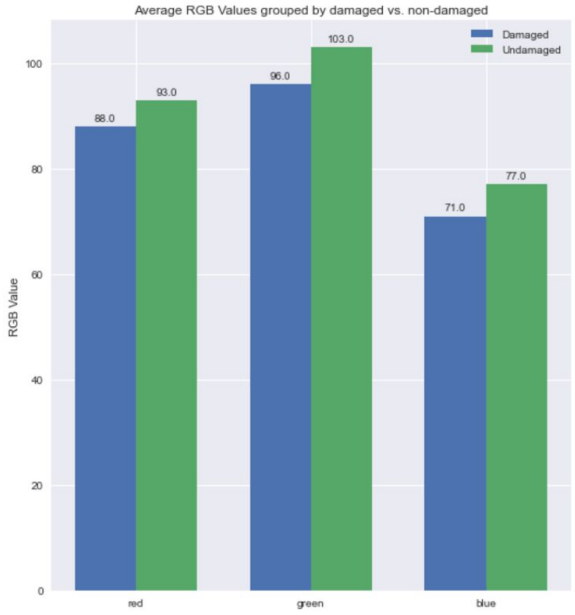

Figure 3: comparative bar chart of mean RGB values for undamaged (green) and damaged (blue) images.

The mean and standard deviation can also be used as a feature vector in order to classify images. This method is also a good way to understand the similarity of images in a dataset, as well as, detect outlier images. Figure 4 shows the mean and standard deviation of each color channel separated by class. The undamaged images have more outliers, specifically images with very high intensity levels and a small standard deviation. We can also see the damaged images have more variance overall than the more tightly clustered undamaged images.

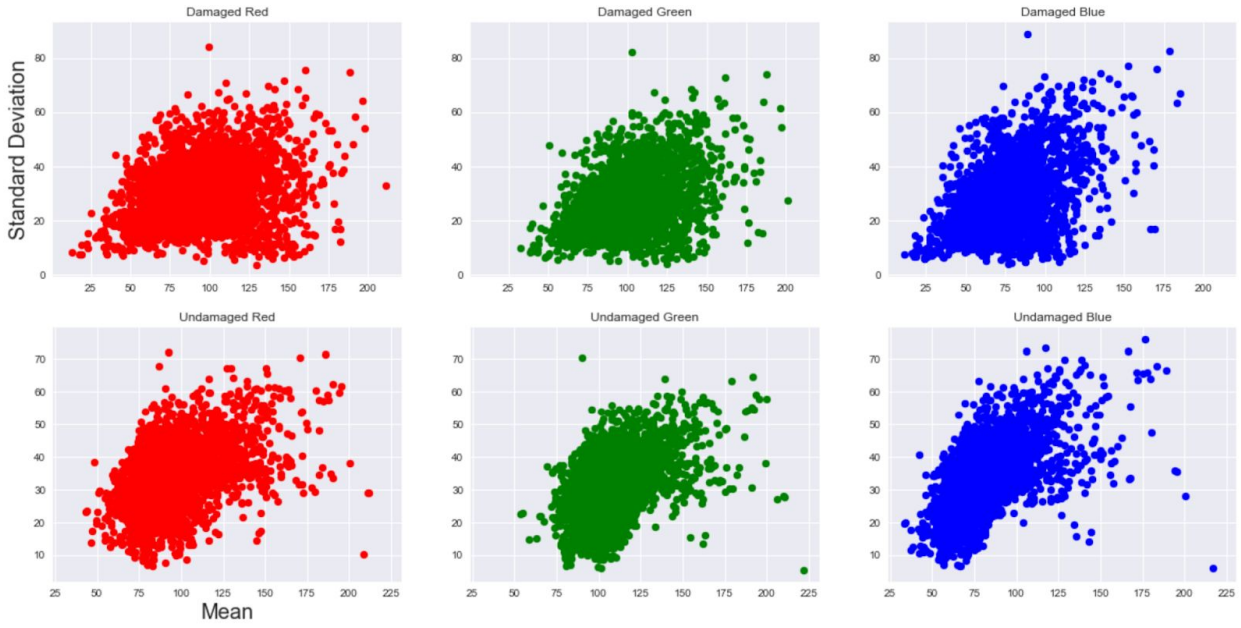

Figure 4: Standard deviation vs. mean in all three color channels. Damaged images are on the top and undamaged images are on the bottom.

We can reach the same conclusion by plotting a histogram of each RGB channel. Figure 5 shows the same trend from the scatter plots. Damaged images have more variance and less intensity overall compared to undamaged images.

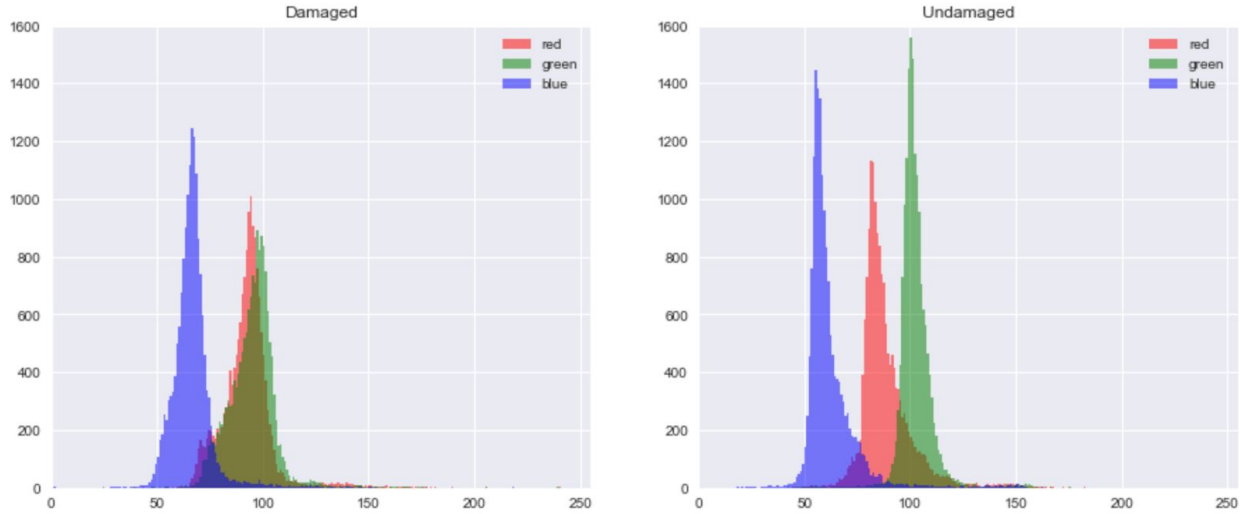

Figure 5: Histogram of RGB values for damaged and undamaged images.

5.3 Mean Image & Structural Similarity Index (SSIM)

The SSIM is used for measuring the similarity between two images. SSIM is often compared to other metrics, including more simple metrics such as mean squared error and normalized root mean squared error. SSIM has been repeatedly shown to significantly outperform MSE in accuracy. The SSIM returns a value between -1 and 1: 1 where the two images are identical and 0 if the images have no structural similarity.

First, we find the “mean image” for each class and calculate the SSIM between those two images. Distinguishing the mean images by eye can be very difficult (Figure 6). The SSIM calculation gave us an index score of .98, agreeing with our human eyes. The damaged images appear to have a slightly more green tint than the non-damaged images. It's possible the damaged images have more of a green tint to them because flooding more consistently occurs near rivers, where the banks are usually covered in green foliage. The prevalence of grayish-brown in the non-damaged images could be due to the gray color of the pavement. When flooding occurs the pavement is most likely covered by the brownish flood water. The

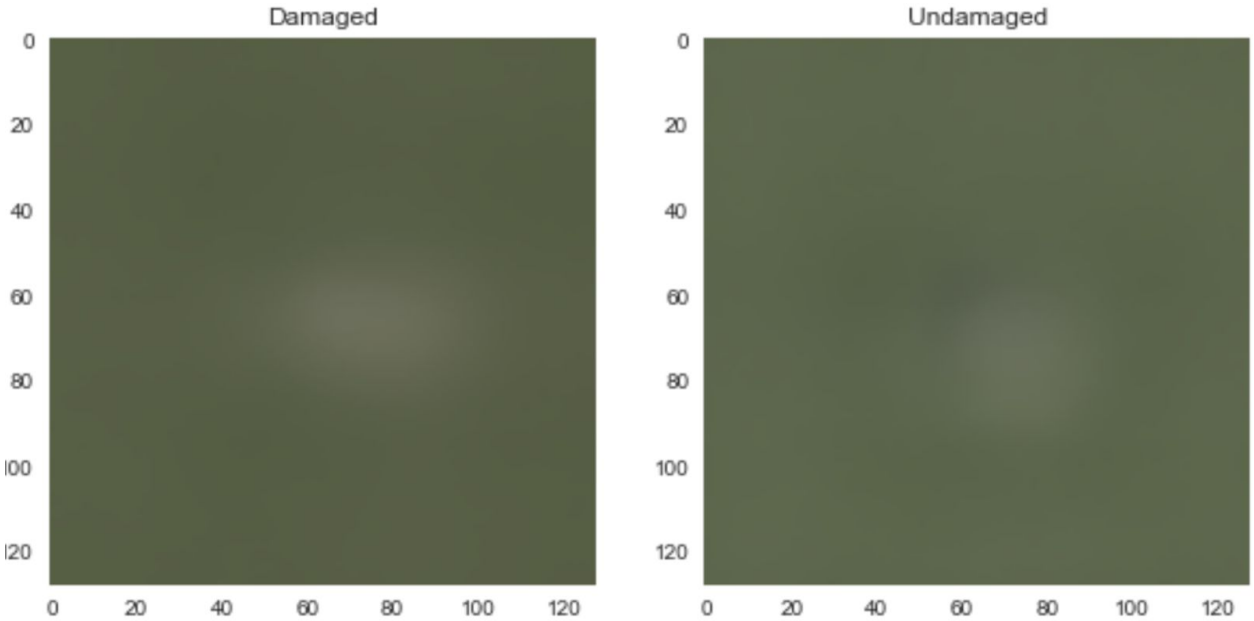

Figure 6: Mean image of damaged and undamaged images.

mean image for both classes are very similar. How could a model possibly predict the difference between the two classes? One possible solution would be identifying different objects in each class and comparing them.

5.4 Object Detection

Object detection is a computer vision technique that allows us to identify and locate objects in an image. It is yet another feature that can be used to distinguish between classes. First we use image thresholding, perhaps the most simple form of object detection. Image thresholding replaces each pixel in an image with a black pixel if the image intensity is less than 128 or a white pixel if the image intensity is greater than that 128. Figure 7 shows two images on the shore of a body of water. The damaged image has debris scattered across the water. The threshold filter does a good job of separating the water and buildings in the images for the most part. It has a hard time separating water from grass as you can see in the non-damaged image. It also completely misses the large debris in the middle of the damaged image. We can use edge detection, a more sophisticated approach, to segment these images.

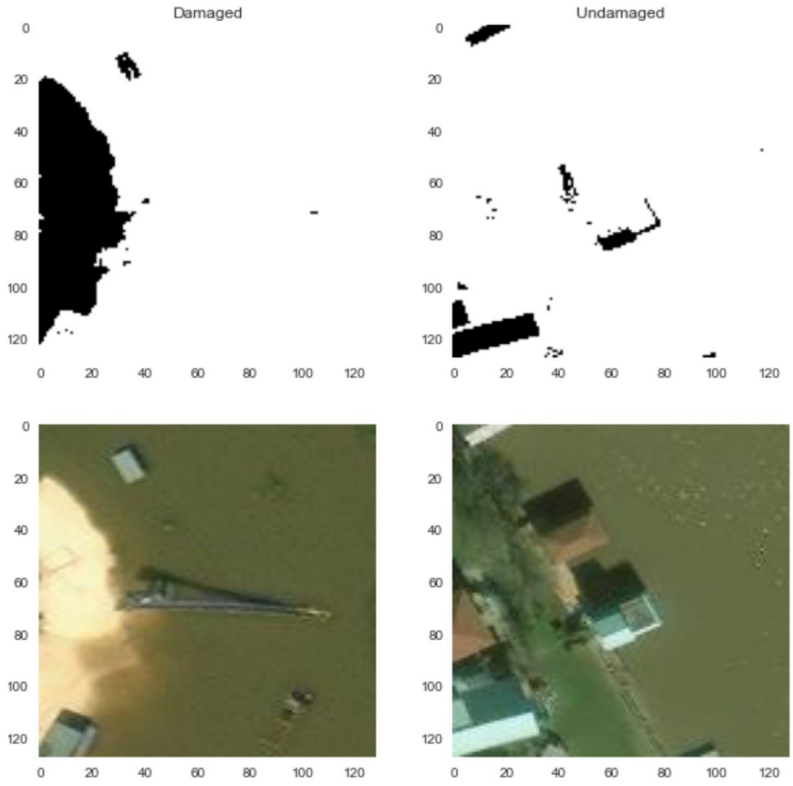

Figure 7: Original image (bottom) compared to image after thresholding (top).

Edge detection is used when we want to divide the image into areas corresponding to different objects. The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images.

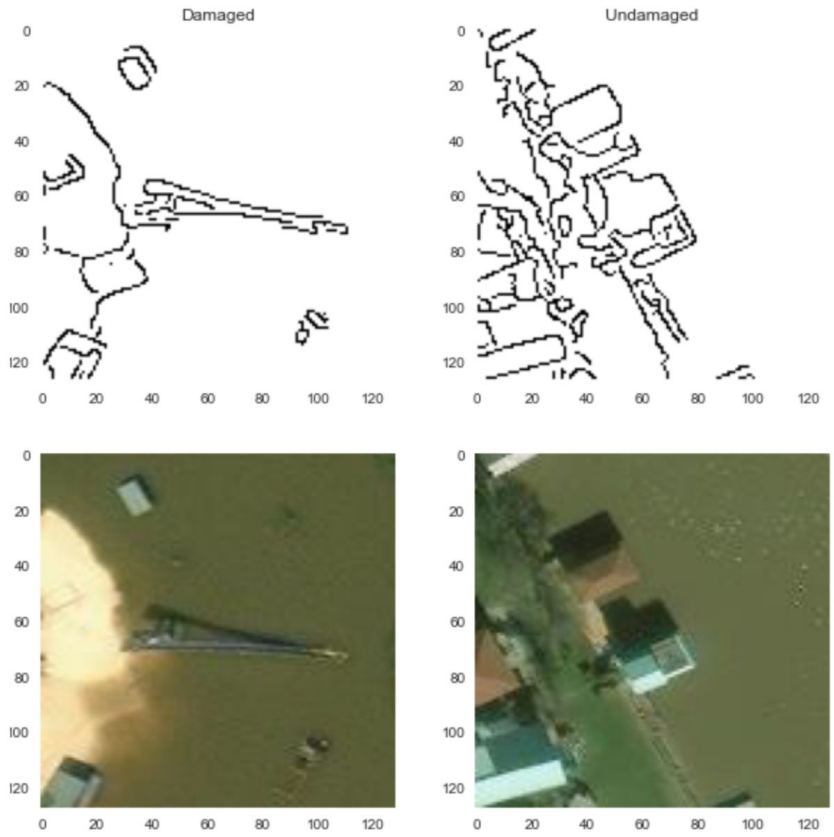

Figure 8: Original image (bottom) compared to image after using Canny edge detector (top).

The canny edge detector does a better job of detecting buildings and debris from the images. This time, the large object in the middle of the damaged image was recognized. The canny images leave whitespace for water which could be useful later on during modeling. It's possible a neural network could use these features to differentiate between the two classes.

For the case of thoroughness we'll use Histogram of oriented gradients (HOG) for object detection. HOG is a feature descriptor used in computer vision and image processing for the

purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image.

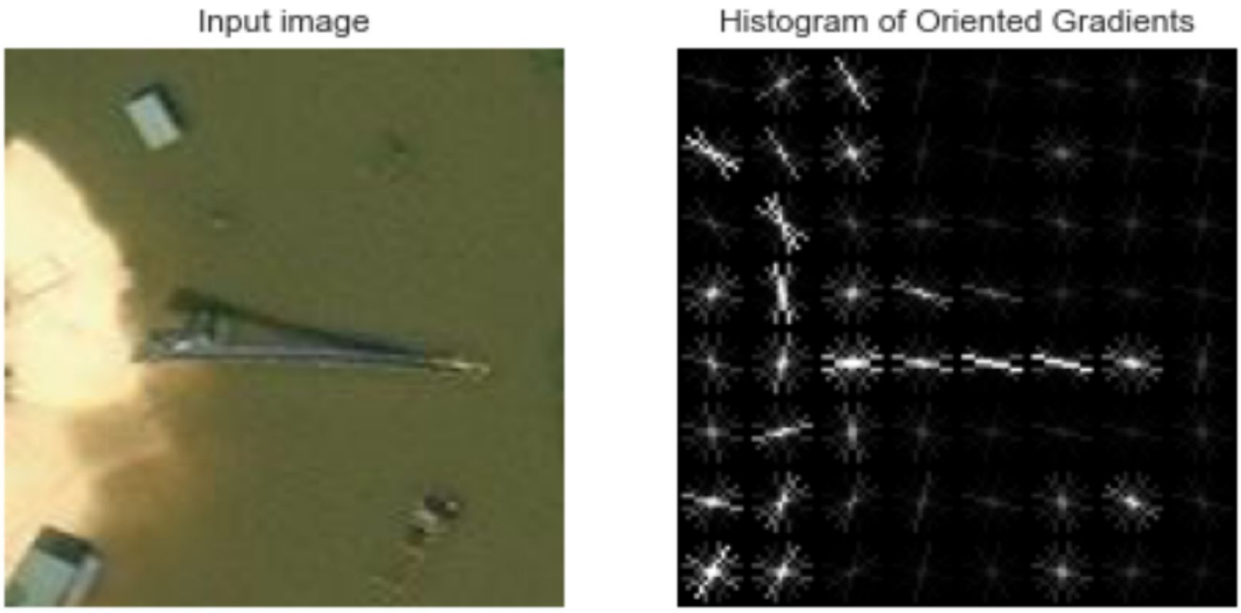

Figure 9: Original image (left) compared to image of HOG transformed image(right).

The HOG technique does a good of detecting objects as well. It manages to capture smaller objects like the tree in the middle of the water (Figure 9).

We were able to extract a fair amount of features from our EDA. The two classes of images are in general very similar structurally and color-wise. The undamaged images have slightly more intensity in all three channels which leads to the damaged images having more of a green tint. The damaged images have more variance in the distribution of means for all three rgb channels. Lastly, the two classes appear to be differentiated by object detection, specifically detecting debris in the damaged class. Next, we'll build a CNN in hopes of predicting these two classes accurately.

6 Modeling

I chose to work with the Python's Keras library for training and testing the neural network. I used a number of metrics to evaluate the model including precision, recall, and f1 score. Because we are dealing with an unbalanced test set, accuracy is not the best metric for success.
Before modeling, the training images were augmented through random rotation, horizontal flip, vertical and horizontal shift, shear, and zoom to avoid overfitting. Hopefully this will lead to better generalization and achieve better validation and test accuracy.

6.1 Building the Convolutional Neural Network

The convolutional neural network (CNN) often yields outstanding results over other algorithms for computer vision tasks such as object categorization, image classification, and object recognition. Structurally, CNN is a feed-forward network that is particularly powerful in extracting hierarchical features from images. The common structure of CNN has three components: the convolutional layer, the subsampling layer, and the fully connected layer.

To build our model we use a number of convolutional layers with a 3x3 filter followed by a max pooling layer until we reach a fully connected layer. In our first layer, we have 32 filters that represent 32 ways to extract features from the previous layers and form a stack of 32 feature matrices. The max pooling layer reduces the input feature matrix to half its number of columns and rows, which helps to reduce the resolution by a factor of 4 and the network's sensitivity to distortion. After the features are extracted and the resolution reduced, we add a flattening layer that will transform the matrix into a feature vector. We then add a dropout layer that prevents overfitting. Dropout prevents neurons from remembering too much training data by dropping randomly chosen neurons. Finally, we feed this vector into a fully connected layer in which predictions can be made.

For activation functions in the CNN, a rectified linear unit (ReLU) is a common choice. ReLU will output the input directly if it is positive, otherwise, it will output zero. The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better. For the fully connected layer we use a sigmoid function in order to produce a probability of each class.

6.2 Model Performance

We train our model over 100 epochs and evaluate its performance. The model performs exceptionally well without showing signs of overfitting. The model doesn't show many signs of overfitting as our validation accuracy follows the training accuracy very closely throughout the training epochs (Figure 10).

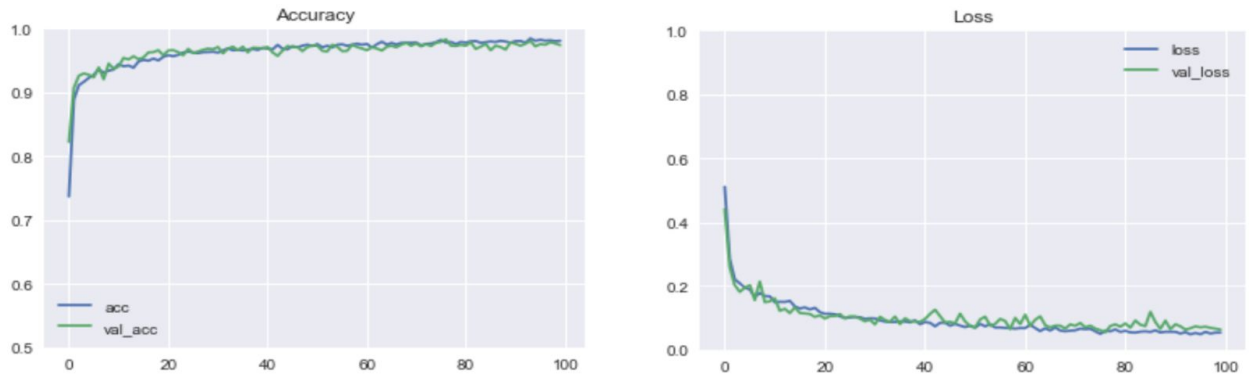

Figure 10: Accuracy & loss over training epochs.

The model performs fairly well on both the balanced and unbalanced test sets. In contrast to the balanced test set which has a baseline accuracy of 50%, the baseline accuracy for the unbalanced test set is 88.89%, which can be achieved by annotating all buildings as the majority class, damaged. So to see an accuracy of 98% is a marked improvement. Since recall is higher than precision, our model is better at finding members of the positive class than

correctly classifying members of the positive class. This means our model has very little false negatives and the classifier is more permissive in the criteria for classifying something as positive. Table 1 summarizes the performance of the model.

Metric	Training	Validation	Balanced Test	Unbalanced Test
Loss	0.04	0.06	0.04	0.04
Accuracy	0.98	0.98	0.99	0.98
F1 Score	0.98	0.98	0.99	0.92
Recall	0.99	0.99	0.98	0.97
Precision	0.98	0.97	0.99	0.89

Table 1: Summary of model metrics.

For good measure, we evaluate our model based on the ROC curve and a precision-recall curve for the unbalanced set. Our model performs exceptionally well on both the balanced and unbalanced sets according to the AUC score (Figure 11).

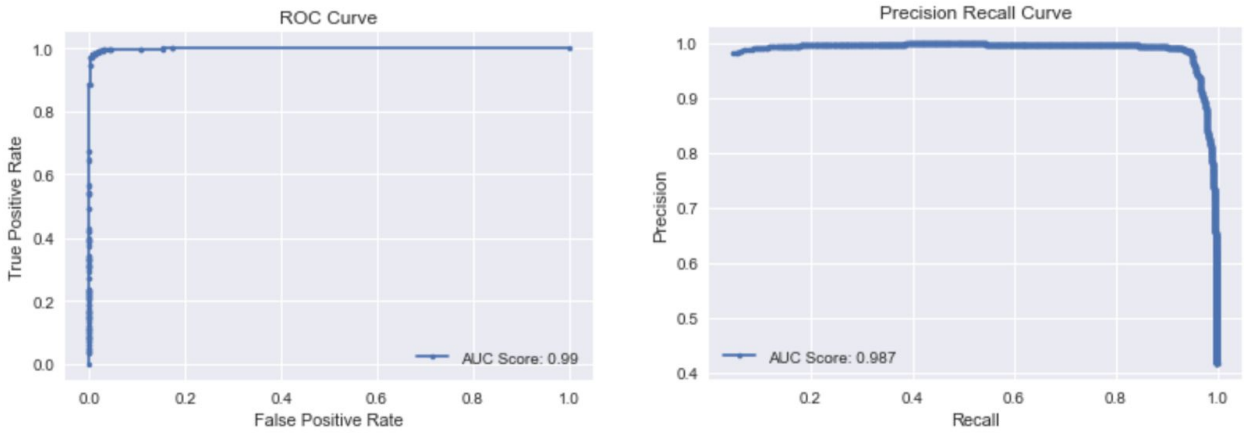

Figure 11: ROC curve (left) & precision-recall curve (right) for our CNN.

7 Misclassified Images

Although our model performed exceptionally well it's important to Investigate instances where the algorithm makes wrong classifications to see if any intuition can be derived. Figure 12 shows the false positives (images that were predicted as damaged but were actually undamaged) and Figure 13 shows false negatives (images that were predicted as undamaged but were actually damaged). Out of 1000 possible undamaged images the CNN classified only 27 incorrectly. We hypothesize that the algorithm could predict the damage through flood water and/or debris edges. Under such a hypothesis, the building in the center of image 16, the water in image 4 and 22, the cloud covering in image 18 and 27 and the parking lot with scattered cars in image 12 can potentially mislead the model. The model seems to misclassify scattered cars or "junk" in people's yard as debris.

Figure 12: False positive images.

Out of the possible 8000 damaged images the CNN classified only 115 incorrectly. The false negative cases are harder to make sense out of the prediction. Even through careful visual inspection, we cannot see image 8 as being flooded/damaged. This image, in particular, is more blurry than the rest of the images. It could potentially be labeling mistakes by the volunteers due to the blurriness. On the other hand, many of the images are clearly flooded/damaged, but the

algorithm misses them. In the case of images 24, 57, 76, 88, 101, and 112 the cloud cover might be misleading the model as we saw before. We see the algorithm misclassify the baseball diamond in images 28, 31, 65, and 74. Perhaps this is because of the unique structure of the diamond. Although these images don't look flooded or damaged in my eyes, perhaps it's a labelling mistake as well.

Figure 13: False negative images.

8 Interpreting the Neural Network

Convolutional Neural Networks have led to unprecedented breakthroughs in a variety of computer vision tasks in the last few years. Our CNN like many others perform exceptionally well on computer vision tasks, but their lack of decomposability into intuitive components makes them hard to interpret. Interpretability of Deep Learning models is essential for stakeholders who want to understand what features are used in distinguishing between classes. Model transparency is useful to explain why they predict what they predict.

8.1 Visualizing Activation Layers

One way to achieve model interpretability is to visualize activation layers of a CNN. Using the python library Keract, we can gain a better understanding of how our model is making predictions in each layer. Visualizing activation layers consists of displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input. This gives a view into how an input is decomposed into the different filters learned by the network. Figure 14 &15 show the activations of a single undamaged and damaged image respectively. The initial layers act as a collection of edge extraction. At the last layer, the information is more abstract and less visually interpretable. In the damaged image activation layers we again see the initial layers extract edges and the last layer extract smaller areas of the image. These smaller areas of pixels highlighted could possibly be debris the model is picking up. Since we see more smaller areas highlighted in the damaged image, this might be one way the model differentiates between classes.

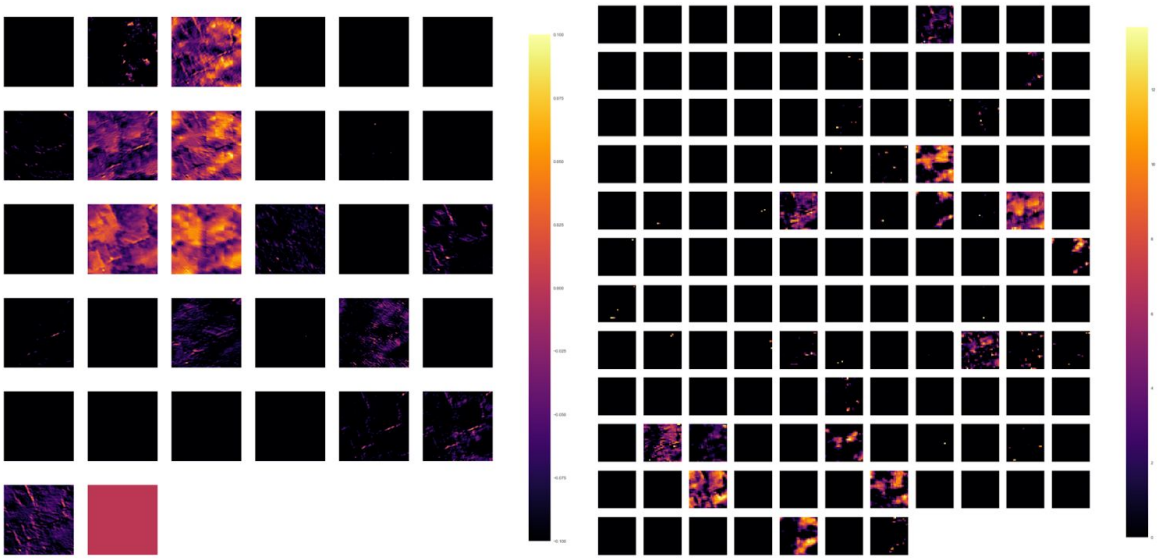

Figure 14: Visual representation of activation layers of a single undamaged image. First activation layer is on the left and third activation layer is on the right.

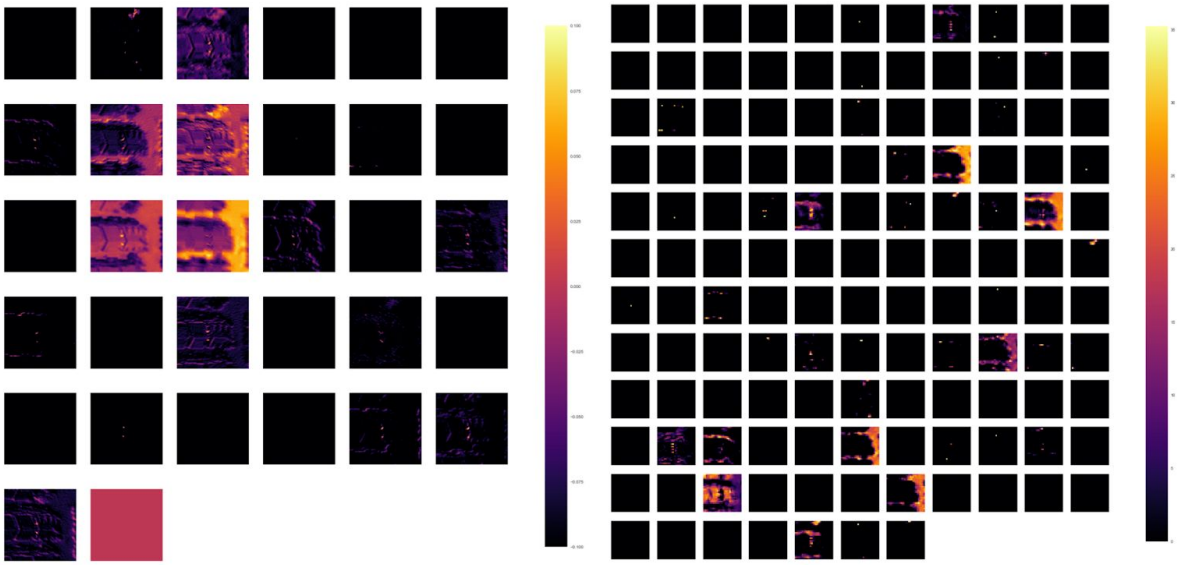

Figure 15: Visual representation of activation layers of a single damaged image. First activation layer is on the left and third activation layer is on the right.

8.2 Grad-Cam

Another way to interpret neural networks is visualizing the gradients of images to gain a better understanding of what features are being selected. Grad-Cam uses the gradient information flowing into the last convolutional layer of the CNN to understand each neuron for a decision of interest. The GRAD-CAM images show exactly what the model used to determine each class in the form of gradients. Darker gradients indicate more important features used to determine class in each image. Figure 16 shows the original image and its labelled compared to the gradient of that image when fed through the neural network. It's immediately apparent the damaged images gradients differ from the undamaged gradients. The images classified as damaged almost all highlight edges in the form of buildings or debris followed by very low gradients surrounding the buildings. The model has very low gradients for flood water compared to buildings/debris. In contrast, the undamaged images have a more evenly distributed gradient profile across the whole image. In the case where the model

misclassified an undamaged image for a damaged one, we see that same trend, a large highlighted gradient surrounded by very low gradients. This trend is one possible explanation for how the model differentiates between the two classes.

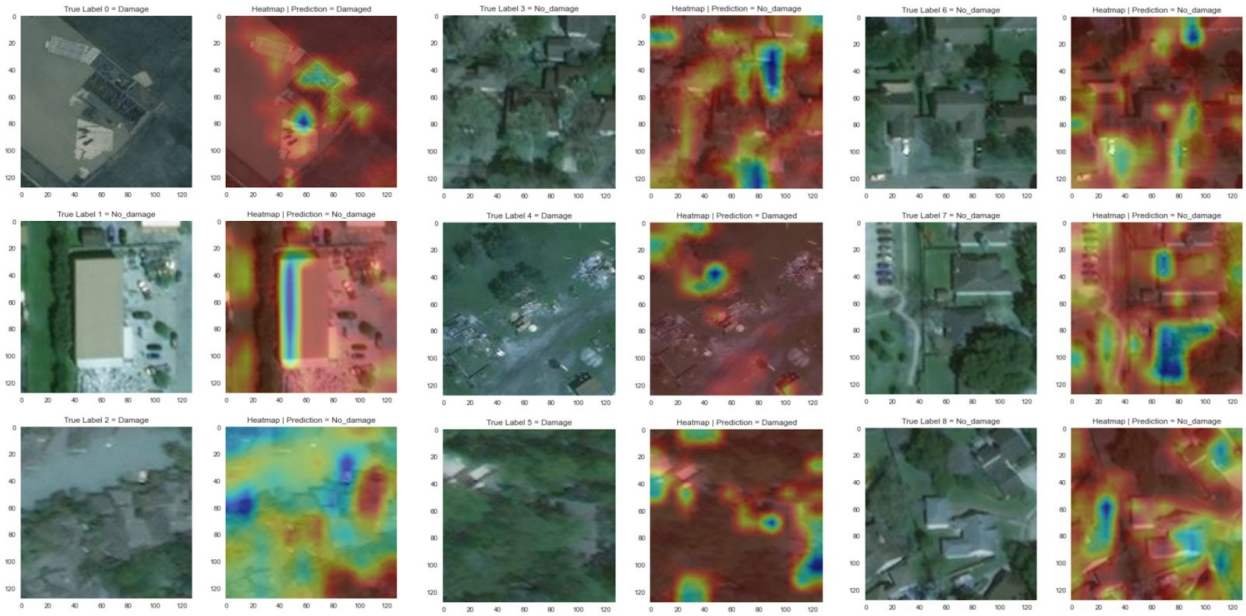

Figure 16: Original image and its true label compared to highlighted gradients and the prediction label.

9 How does the model generalize to new data?

Because our images are all from the same source already preprocessed, it would be interesting to know if our model can predict accurately with random images that haven't been carefully selected. We will pick satellite images in locations that could potentially be affected by a hurricane and make predictions on those images. Because it's hard to select damaged images from satellite data, each image will be undamaged. If our model is any good, it should be able to generalize to these new images. Out of the 55 images 8 were misclassified as damaged. Although this isn't quite as impressive as the test images, it's still inspiring to see the model was able to generalize to new images.

Figure 17: Model misclassifications from new images.

Again, It's hard to tell what the model is looking at in figure 17 but the common theme is the presence of a brownish color surrounding the images. That same brown color could be used to identify flooded areas. With this limited dataset it would be impossible to pin down exactly where the model trips up but we did learn two things:

1. Our model generalizes well to new images but prediction performance still drastically decreases.

2. In order for the model to generalize better we need more images from varying locations affected by hurricanes.

10 Conclusion & Future Research

We demonstrated that convolutional neural networks can reliably classify damaged buildings on post-hurricane satellite imagery with high accuracy. Although it's difficult to tell exactly how the CNN is classifying each image, it's most likely a combination of features outlined above including: color distribution, mean image & SSIM, and edge detection.

Although this data is limited to the properties of the Greater Houston area during Hurricane Harvey, the model can be further improved and generalized to future hurricane events in other regions by collecting more damaged and undamaged samples from past events and areas. More data from other areas would hopefully lead to a more robust model that can generalize better to new data.

For faster disaster response, the model should be able to classify satellite images from lower quality images including images with clouds obscuring part of the land. Also, there might not be enough time to pre-process images well due to the gravity of the emergency situations hurricanes are likely to cause. It would be advantageous to further investigate how the model could adjust to more noise in the data due to different sizes and zoom levels of each image. It would also be helpful to extend the model to classify road damage and debris, which could help disaster relief workers plan transportation routes for medical supplies, food, or fuel to hurricane survivors.

References

Cao, Q. D., & Choe, Y. (2018). Building Damage Annotation on Post-Hurricane Satellite Imagery Based on Convolutional Neural Networks. *IEEE Dataport*. https://dx.doi.org/10.21227/sdad-1e56