Final Project

Ryan

December 14, 2020

Abstract

A simple K-Nearest-Neighbor algorithm to attempt to accurately classify diseased trees from a set of diseased and non-diseased foliage. The samples include both trees and surrounding plants and landforms according to the description given from the UCI (University of California, Irvine) repository.

Introduction

This dataset[2] defines attributes for classifying wilted or diseased trees and was obtained from satellite imagery. It contains data for both diseased trees and surrounding foliage in an attempt to differentiate between them. I did not see a specific goal listed from the abstract of their paper, but I can assume several benefits from being able to classify diseased trees from satellite imagery. With an ever changing environment and influence from humans, being able to monitor the health of the surrounding ecosystem would be beneficial in heading off any damage associated with either. It would also be beneficial to understand the spread of disease through trees and foliage.

Data Set

This data was obtained from the high-resolution remote sensing (Quickbird) satellite and modified by Johnson B., Tateishi, R., and Hoan, N. into the attributes, GLCM (Gray Level Co-occurrence Matrix), mean green value, mean red value, mean NIR (Near-Infrared) value, and standard deviation of the panchromatic band. The original data was taken in the panchromatic bands of the spectral imaging hardware on the quickbird satellite. The attributes in the data set are listed as follows:

- GLCM_Pan: A technique to attempt to differentiate the "texture" of images by sampling pixels and their neighbors. It works, in essence, by counting the occurrences of pixel pairs. The end result is an MxM matrix where M is the largest pixel value plus one. Each cell holds the number of occurrences of each pixel pair where the row and column are the two pixels values. In this data set, the GLCM mean is the value used for the GLCM_Pan attribute. The mean is given by the following equations[3, p.41]:

  $$\mu_i = \sum_{i,j=1}^{N-1} i(P_{i,j}) \qquad \mu_j = \sum_{i,j=1}^{N-1} i(P_{i,j})$$

  In both of these equations, the i represents the reference pixel and the j represents the neighbor pixel. $\mathrm{P_{i,j}}$ is the probability of the reference value occurring with the current neighbor pixel, j, and N is equal to the largest pixel value plus one. The GLCM is usually symmetric so both of these equations are usually equal.

  More information can be found here: `https://prism.ucalgary.ca/bitstream/handle/1880/51900/texture%20tutorial%20v%203_0%20180206.pdf`

- Mean_G: The mean value of green color from the image.
- Mean_R: The mean value of red color from the image.
- Mean_NIR: The mean value of Near-Infrared from the image.
- SD_Pan: The standard deviation of the panchromatic band.

I was only able to see the abstract of the paper[1] that produced this data set so I am unable to dig too deeply into what they mean by Mean_G, Mean_R, or Mean_NIR although I assume it has to do with the mean of the colors represented by each pixel. It appears that they used several techniques to produce these attributes but I only briefly touched on them based on what I was able to learn from my own personal research. My main focus on this project was to use a technique I learned from class to attempt to accurately classify elements from the test set using the elements in the training set.

1 Methods

For this project I used the built in function from sklearn, KNeighborsClassifier, to classify the diseased trees. As I did not have a full understanding of some of the algorithms available for the classifier, I stuck with the brute force algorithm. This algorithm has several options to choose from but I will focus on the euclidean distance algorithm as this was the one I learned in class and am the most familiar with. The euclidean distance can be found by using the following formula:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

It takes the x and y values of two points and calculates the straight line distance between them. Because we are looking at several attributes and not just simple x and y, we can adjust our formula as follows:

$$distance = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2}$$

In this situation, N is the number of attributes we are working with, x is the attribute from our test set, and y is the attribute from our training set. We then take an element from the test set and perform this algorithm with each element in the training set. This gives us a list of distances. We then find the training element that has the smallest distance to our test element and guess that our test element is the same class as the training element. This assumes just one nearest neighbor. We can adjust our K number of neighbors to choose the K closest neighbors. If we have more than one neighbor then we find which class occurred the most among those K neighbors and choose that as our guess.

To summarize the process in my code, I first created the KNeighborsClassifier and assigned a K value. I then used it's fit function to fit the training data. Lastly, I used it's predict function to predict each of the elements from the test set and printed out an accuracy rating by comparing the predicted set to the test set's labels.

Details of the KNeighborsClassifier function can be found here: `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`

2 Results

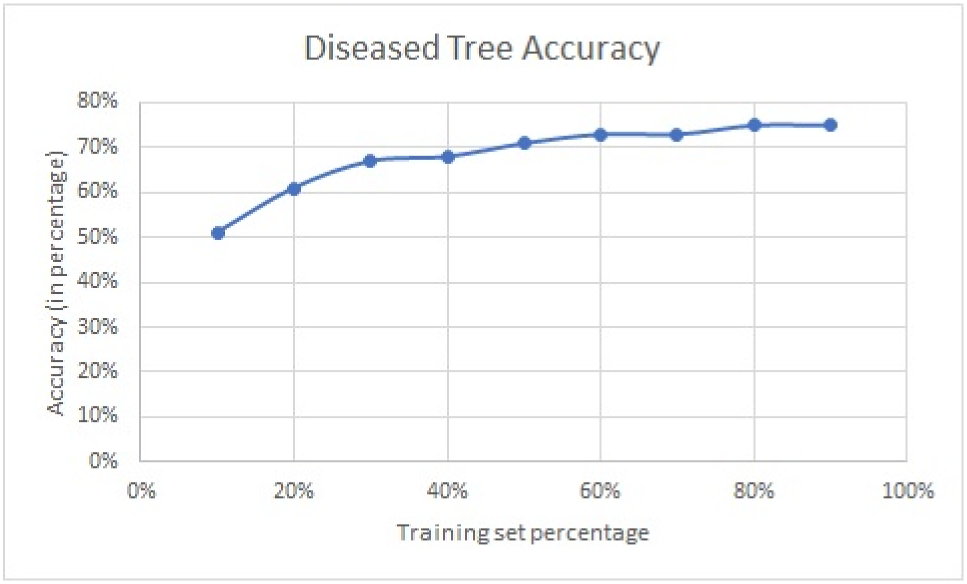
My initial results indicated an 80.8% accuracy rate when using the test/train split that was provided. When I dug into the specifics of the split, I saw that the training set consisted of 74 diseased trees and 4,265 non-diseased foliage. The test set consisted of 187 diseased trees and 313 non-diseased foliage. I found it interesting that there were more diseased tree samples in the test set than in the training set as I thought that we generally wanted to have more sample representations in the training set. When I checked back in the repository I saw that they used stratified random sampling to determine the test set. I produced a confusion matrix for this testing below:

$$\begin{array}{r c} & \begin{array}{cc} Diseased & Non-Diseased \end{array} \\ \begin{array}{r} Diseased \\ Non-Diseased \end{array} & \left[\begin{array}{cc} 98 & 89 \\ 7 & 306 \end{array}\right] \end{array}$$

When calculating the misclassification rate for each type, I found that diseased trees had a 47.59% misclassification rate and non-diseased foliage had a 2.24% misclassification rate. This shows that diseased trees have a high chance of being incorrectly classified as non-diseased but non-diseased foliage has a low chance of being classified as diseased. Because of this, we can safely say that if our algorithm classifies something as diseased, it has a high chance of actually being diseased. Sadly, this also means that if something is classified as non-diseased, there is a decent chance that it is actually diseased.

I was curious what would happen if I modified the train/test split so I set about readjusting the representation of each class in the training and test sets. I combined both the test set and training set and created a new list to hold the labels from them and then made use of the built in method, train_test_split from sklearn. This method takes in a data set and labels and creates a training and test set along with two lists for the labels. It also gives you the ability to modify the test size by a percentage. I found that the method was good at equally splitting both classes by the percentage. I then played around with the numbers to see how it would affect the accuracy. My results indicated best accuracy at a 10% test split. My diseased tree hit accuracy graph and confusion matrices are given on the next page.

Accuracy as we increase the training set's share of the data:



Confusion matrices as we increase the training set's share of the data: (Note: confusion matrices follow the same labeling as the matrix on page 4)

$$10\% : \begin{bmatrix} 121 & 115 \\ 46 & 4075 \end{bmatrix} \qquad 20\% : \begin{bmatrix} 129 & 81 \\ 31 & 3631 \end{bmatrix} \qquad 30\% : \begin{bmatrix} 121 & 62 \\ 26 & 3179 \end{bmatrix}$$

$$40\% : \begin{bmatrix} 108 & 48 \\ 22 & 2727 \end{bmatrix} \qquad 50\% : \begin{bmatrix} 94 & 38 \\ 17 & 2271 \end{bmatrix} \qquad 60\% : \begin{bmatrix} 77 & 28 \\ 13 & 1817 \end{bmatrix}$$

$$70\% : \begin{bmatrix} 57 & 21 \\ 10 & 1364 \end{bmatrix} \qquad 80\% : \begin{bmatrix} 40 & 13 \\ 6 & 908 \end{bmatrix} \qquad 90\% : \begin{bmatrix} 20 & 6 \\ 3 & 455 \end{bmatrix}$$

As we increase the training set's share of the data, we increase the accuracy of our diseased tree detection. It starts out at roughly 50% accuracy when we have a training size of 10% of the data. It gradually increases to about 75% when we have a training size of 90% of the data. The graph is only for the diseased tree class. The non-diseased class has excellent accuracy, increasing to greater than 99% as we increase the training set's share.

We only have 261 diseased trees to work with in this data set so I believe the accuracy would be even greater if we had more samples to work with.

The overall accuracy of the algorithm increases to about 97.5% when using the train_test_split method from sklearn. I realize that part of this is due to the large presence of non-diseased foliage and the low amount of diseased tree samples. Regardless, I believe that K-Nearest-Neighbor is a decent option when attempting to classify this data set with the attributes given. It needs additional tweaking in order to function at its best, but I am happy with the results I have managed to obtain, even with a simple nearest neighbor approach.

3 Discussion

3.1 Decision 1

When approaching this project, I was unsure of what I wanted to do. I sought out different data sets and hoped to find ones that I felt confident I could tackle with what I have learned up to this point. I found this data set and thought it was interesting that they had taken data from satellite imagery and turned it into a few key attributes and wondered if I could use nearest neighbor to get an accuracte classification of the elements. I had created a simple nearest neighbor algorithm and was able to plug this data set into it. After a few tweaks I found that it returned an accuracy of 80.8%. This peaked my interest so I delved further into the data set to determine how it was set up and to see if I might be able to increase the accuracy even further.

After looking through the data, I realized that they had more samples of diseased trees in the test set than the training set. I wondered what would happen if I modified that, so I got to work on it. I struggled a bit at first because I needed to figure out a way to combine the two sets and then split each class roughly equally into a new train and test set. My first iteration had me grabbing every diseased class and putting it into a list and then grabbing every non-diseased class and putting it into another list. Then I split each one based on a percentage and recombined them into a new train and test set. This worked but I wondered if there was another more succinct way to do this so I searched online for a different way.

I found the train_test_split function from sklearn and it worked perfectly. All I had to do was to split the labels from the attributes and then let the function do the rest. This cut down on a lot of code which I was happy about.

3.2 Decision 2

I then decided that I wanted to do a comparison of the original train/test split and my own. I split up the code to reflect this, which was rather easy to do. I basically just needed to run the algorithm on both sets separately. At this point I was using a simple euclidean formula to calculate the distances and the time it took was noticeable. I knew there were libraries out there that could accomplish what I wanted with less visible code and in less time.

I searched online and found the KNeighborsClassifier function from sklearn. After researching how it worked, it was a simple matter to tweak my original code to include it. Once I implemented it, I was amazed at how much faster it accomplished the calculations. It would calculate the distances and the predictions in less than a second and I was able to test my program more quickly thanks to it. I was also happy to see that the results it was returning were the same as what I was accomplishing with my original approach.

3.3 Difficulties

One difficulty I encountered was understanding the attributes associated with the data set. The main one, GLCM, took me some time to research and understand before I felt confident enough to write a basic description of what it was. The other attributes were fairly self-explanatory, but the GLCM attribute required more effort to understand. I spent a couple of hours reading a pdf and other sources online until I had a decent understanding of it.

Another difficulty I encountered was inconsistent results when testing the confusion matrix. My results were not making sense as I changed the train and test split percentage. My diseased trees hit rate was increasing as I removed diseased classes from the training set, until I was hitting greater than 99%. I knew that this could not possibly be correct so I took a look back at my code to figure out what was going wrong. I realized that I had my testing criteria backwards when checking for misclassified classes. For example, if my algorithm predicted a diseased tree but in reality it was non-diseased foliage, then I should have added that to the incorrect foliage classification numbers. Instead, I was adding it to the incorrect diseased tree classificiation numbers.

4 What I've learned

During this project, I learned that I was beginning to have a much better understanding of what goes into machine learning and how to approach some data sets. For a large portion of the class I felt that I was not fully understanding how it all fit together, but as I worked with several data sets I started to have a better understanding of how the algorithms related to the data. I also gained a better understanding of how to approach libraries online in order to fit them into my own programs and to successfully make use of them.

I believe I have also gained a greater appreciation for the fun in attempting to write algorithms to classify data. I had fun writing the code for this project and in finding ways to increase or optimize the accuracy and runtimes. I want to delve deeper into algorithm optimization in order to make my future programs more efficient and faster.

References

[1] Johnson B, Tateishi R, and Hoan N. A hybrid pansharpening approach and multiscale object-based image analysis for mapping diseased pine and oak trees. *International Journal of Remote Sensing, 34*, (20):6969–6982, 2013.

[2] Johnson B, Tateishi R, and Hoan N. Wilt data set, 2013.

[3] Hall-Byer and Mryka. *GLCM Texture: A tutorial.* University of Calgary, Calgary, Alberta T2N 1N4 Canada, 2017.