SpecAI.Seg Details

Scout Jarman, Zac Rios, Eric Aikens, Kyle Steinicke

Introduction

This document is a centralized place to describe all the desired functionality for the SpecAI.Seg package, along with who will be working on each part. Each section will detail a main component of the code to be implemented, including the desired functionality/structure of the function so that everything can work together. There will be basic functionality which will be a minimum requirement. I will also include bonus functionality for after the basic functionality is completed.

For simplicity, we will have everyone document their own function, and to write their own test cases. At the very least the test cases should verify error checking, but other functions will be easier to make proper checks for. Also when we get to writing the vignette, everyone will be primarily in charge of writing a section on how to use your functions.

Note that this package is to act similarly to the `SpecAI.Seg` package in Python. The code can be found here, so please take a look at the read me (and some of the actual functions) to see the general idea for how this package should work.

Data.R

Author: Kyle Steinicke

This file will hold all of the code for loading in the data, formatting it appropriately, and creating a custom S3 class called "HSI_data". This is the first function that needs to be created, as no one else can work on their code until this is completed. You can take a look at the get_data function to see exactly how it is done in Python.

Details

There are a set of 6 standard hyperspectral images that can be downloaded from the internet. I have created an internal dataset called image_details, which is a list containing all the necessary information for downloading these 6 images. The 6 images are called Indian Pines (image_details$IndianPines), Pavia Center (image_details$PaviaC), Pavia University (image_details$PaviaU), Kennedy Space Center (image_details$KSC), and Botswana (image_details$Botswana). Each of these are also a list (that is, image_details is a list of lists), with the following additional named items:

- `urls`: A vector of two strings containing the urls for the image, and the ground truth associated with the image.
- `img`: The file name of the file that contains the image.
- `img_key`: The URL downloads the image as a `.mat` file (like from matlab). When this data is loaded into R (using an appropriate function to open a .mat file), it returns a names list. img_key is they key to get the image from that list.
- `gt`: Like img, but for ground truth.

- `gt_key`: Like img_key, but for the ground truth.
- `rgb_bands`: The three channels to correspond to the Red Green and Blue channels for plotting the actual image. When the image is loaded in, it is in the shape of an X by Y by C array, where X and Y represent the spatial dimensions, and C represents the spectral dimensions. For example, Indian Pines is 145 by 145 by 200, meaning this image has 145 by 145 pixels, where each pixel is a 200 length vector. Because we can only plot 3 channel image, we use rgb-bands to define which channels represent the rgb bands. For example, Indian Pines has rgb_bands of 44, 22, 12. This means the red channel is img[, , 44], the green channel is img[, , 22], and the blue channel is img[, , 12], where img is the array loaded in from the img file.
- `label_values`: The ground truth is a matrix of shape X by Y, where each pixel has a label. label_values is a vector describing what each label (which is stored as an integer) corresponds to in real life. For example, if gt[1, 1] = 2 (i.e. the ground truth for pixel at location 1, 1), then the human readable label is label_values[2], which for Indian Pines is "Alfalfa". It should be noted that the ground truth start labeling at 0! So you will want to add 1 to the ground truth so that the index of label_values matches the integer labels in the ground truth.
- `ignored_labels`: In some cases there are labels that are not of interest, these are stored in ignored_labels. Most of the images have ignored_labels = c(0), meaning the first label_values is an ignored label. This won't be important for this version of this package. Note that like the above, you will want to add one to each value in ignored_labels so that is matches the gt, and the indices for label_values.

Functions

There is one primary function to be exported from this file (you can write more if you like). The function is `get_data(name)`, where name is the name of the dataset you want to download/load. This function should the name of the dataset, check to see if it is one of the image available in the image_details dataset. If it is, the use the appropriate attributes from image_details to download the appropriate files from the web.

Once the files have been downloaded, there is some formatting to do to the image (the file downloaded from key `img_key`). We want the final returned list (as an S3 class) to contain the following keys:

- `img_raw`: This is the array of the image with no preprocessing.
- `img_clipped`: Because there can be outliers in the raw data, we do what is called clipping the image. This works by finding the .25 percentile and the 99.75 percentile of all values in the array (ignoring all dimensions). Then for all value in the image that are above 99.75, the values are set to be equal to the 99.75 percentile. Similarly, for all values in the image that are below .25 percentile, the values are set to be equal to the .25 percentile.
- `img`: This image is the 0-1 scaled version of `img_clipped`. A 0-1 scaling means that the largest value in the image is given the value of 1, and the smallest value of the image is set to 0. In pseudo code, this is found by finding (img - min(img)) / (max(img) - min(img)). This is the primary image array that is used throughout the rest of the project.
- `gt`: The ground truth array.
- `label_values`: The label values from image_details.
- `ignored_labels`: The ignored labels from image_details.
- `rgb_bands`: The rgb bands from image_details.
- `img_rgb`: The `img` but only containing the three channels from rgb_bands. Something like `img[, , rgb_bands]`.

An example of what this code should look like is this:

```
get_data("indianpines")
get_data("salinas")
get_data("PaviaU")
```

```r
# Capitalization shouldn't matter, so the following should be equivalent.
get_data("indianpines")
get_data("IndianPines")
get_data("INDIANPINES")

# Class should be "HSI_data"
data <- get_data("indianpines")
class(data) == "HSI_data"  # should be TRUE
```

Watershed.R

Author: Scout and Zac

This file should contain all the code for performing Watershed segmentation. There are two main components for Watershed segmentation. The first is the calculation of the Robust Color Morphological Gradient (see here). The second is the actual running of the Watershed transformation. See the Watershed class in `SpecAI.Seg` Python package.

Details

The first component is calculating the RCMG. Scout will do this using RCPP, and maybe parallel processing depending on computation speed in R.

The second component is calculating the actual watershed segmentation. This is typically done using the `watershed` function from the skimage package. However our goal is to find an R implementation of this algorithm (or if you are feeling adventurous, implementing your own watershed segmentation algorithm). We want to implement the marker based watershed algorithm. Markers are essentially just the coordinates where you want the watershed transform to start. In Python, these are represented as a matrix of all zeros, except where you want the markers to be located. For example, Indian Pines is 145x145 pixels, so you would make a markers array of size 145x145 which is all zeros except for the marker location. For example if you want a marker at pixel [15, 67], then the value of markers[15, 67] should be non zero (typically you can just increment starting from 1). For this project we will just use random marker placement. So the input argument will be the number of markers you want to have (and a random seed for reproducibility), and we will randomly generate those markers.

Functions

There are two main functions. The first is `get_grad(data)`, where data is a `HSI_data` object. This function will return an X by Y matrix representing the gradient of `data$img`. This function will also add an attribute to data called grad which will store the result of the gradient, so that the user doesn't need to keep track of the gradient.

The second function is `watershed(data)`, where data is of class `HSI_data` after being run through get_grad (or if it doesn't have $grad, then run it though get_grad). It should return an X by Y matrix containing integers representing the labels of the segmentation. For example if the watershed segmentation produces 250 segments, then the matrix should have integers ranging from 1 to 250. The value of each pixel in the resulting seg says which segment it is part of. The results will be a matrix, but also be of class `HSI_seg`.

An example of how this code should work:

```
data <- get_data("IndianPines")

get_grad(data)
seg <- watershed(data)

class(seg) == "HSI_seg"  # Should be TRUE
```

Generics.R

Author: Eric Aikens and Zac

This file will contain various useful generics for the `HSI_data` class. We want a `plot` function, which can plot just the image, the image with a segmentation, or the image segmentation average "colors". Also we want basic summary functions for `HSI_data` and `HSI_seg` classes.

Details

The main set of functionality we want is basic generic plotting, that is overriding the `plot` function for our two S3 classes. The first generic would be `plot.HSI_data(x, y, ...)`, which will take in an `HSI_data` object (like that obtained from `get_data()`) and plot the `img_rgb` attribute when the `y` value is ignored. Some easy options are to use `countcolors::plotArrayAsImage`, or `ggimage`.

The second generic would be `plot.HSI_seg`, which will take in an `HSI_seg` object (like that obtained from `watershed()`). This plot will take an `HSI_seg` as the `x` argument and the `HSI_data` as the `y` argument. The plot will actually be of two plots, one is the plot of the img_rgb picture after having the segment boundaries markers (see `mark_boundaries` next). The other plot side by side is a line plot of the average "color" for each segment in the image. This is accomplished by the following pseudo code:

1. for label in seg
2. select the pixels from img which belong to label
3. find the average for each channel
4. make line plot for each segment average

For example, the Indian Pines image is 145x145x200, meaning every pixel is a 200 length vector. Say we generate a segmentation like `seg <- watershed(data, 1000, 789)`, where we specify that we want to break this image into 1000 segments. Note that with watershed, you won't get as many segments as you specify, so for the example above you may only get say 250 segments. The plot we want to make will therefore have 250 lines, where the x axis is from 1 to 200.

We also need another function to mark the boundaries of a segmentation. This function will be based off the `skimage.segmentation.mark_boundaries()` function. this function will take in the `img_rgb` array, and the `HSI_seg` object and returned an image with segment boundaries marked in yellow. See the skimage function for details to implement (or see if there is an equivalent function in R).

The last option we want is to essentially combine the functionality of the two previous generic plotting functions. This will be under the `plot.HSI_data` function, but will take `x` as the `HSI_data` object, and `y` is the `HSI_seg` object. When these two options are passed in, there will be two plots generated. On the top will be image with the boundaries of the segmentation marked (resulting from `mark_boundaries`). On the bottom will be the line plot like that generated from `plot.HSI_seg`.

Functions

The first function to be written is the `mark_boundaries` function. This will take in an img_rgb, and a seg matrix, then produce an rgb image where the boundaries of the segmentation are marked in Yellow. See the links above for details.

The second function will be `plot.HSI_data`. The first option is when `x` is `HSI_data`, and `y` is `NULL`, then it will just plot the plain img_rgb. If `x` is `HSI_data` and `y` is `HSI_seg`, then create the pair plot containing the marked boundaries of the image, and the line plot containing the average spectra for each segment.

The third function is `plot.HSI_seg`, which simply plots the segment averages as described above.

The fourth function will be `summary.HSI_data`, which will print out basic summary information about the given object, such as the name of the image, the dimensions of the image, the rgb channels, and the gt labels for the image. Similarly there should be a `summary.HSI_seg`, which should display the number of segments, the number of pixels in each segment, the average number of pixels in each segment.

Here are some examples of how the code should look when functional:

```
data <- get_data("indianpines")
seg <- watershed(data, 1000, 789)  # 1000 markers, seed 789

plot(data)
plot(data, seg)

plot(seg)

summary(data)
summary(seg)
```