Introduction_to_SpecAISeg

```r
library(SpecAI.Seg)
```

In this vignette, we will discus the usage of this package for the purpose of hyperspectral image (HSI) segmentation. We will also demonstrate all the main function usage and options. This package can be found at https://github.com/scoutiii/SpecAI.Seg. Note that the gradient calculations can take several minuets to run depending on your computer speed.

Introduction

Hyperspectral images (HSI) are very similar to color images but differ in the number of channels. Gray-scale images contain 1 channel of information, and color images contain 3 channels of information representing red, green, and blue. HSI on the other hand can have anywhere between 100 to 300 channels representing a whole range of different color wavelengths in the electromagnetic spectrum. In R, we simply store them as arrays, where the first two dimensions are the spatial dimensions, and the third dimension is the spectral dimension.

HSI contains a lot of information, but can be difficult to analyze on a per-pixel basis. We can use image segmentation to help summarize the spectral information, but also capture some spatial information. Image segmentation is the process of breaking an image into groups of similar pixels. It has been studied since the 1970s, so there are many image segmentation algorithms. For this package, we implement the Watershed segmentation algorithm, which has been extended to HSI. This package is based on the SpecAI.Seg package in Python, though with less functionality (Jarman, 2022).

The rest of this vignette will first show how we can download different standard HSI, and how we can make some basic plots to visualize these images. Next we will describe how to run the Watershed segmentation algorithm, along with the various parameters that can change the segmentation results. Lastly we will show how to make some basic plots of the segmentation results.

Data Loading

Before segmenting or plotting the hyperspectral images, we must first download and clean the data. This package contains two functions to download and save the HSI data, `get_data()` and `get_all_data()`. The `get_all_data()` function takes no parameters and downloads each of the images in the image_details list. The `get_data()` function downloads a single image in the image_details list. The `get_data()` function saves the image data and outputs an `HSI_data` class object that can be assigned to a variable as shown below where we load in the Indian Pines image.

```r
ip <- get_data("indianpines")
#> Reading in pre-downloaded data...
```

Plotting and Summaries

The variable ip is of class `HSI_data`, allowing one to use generics to show specific information about the ip data set. One can plot the false coloring of the image by simply using the `plot` function. Additionally, one can summarize the dataset features by using the `summary` function.

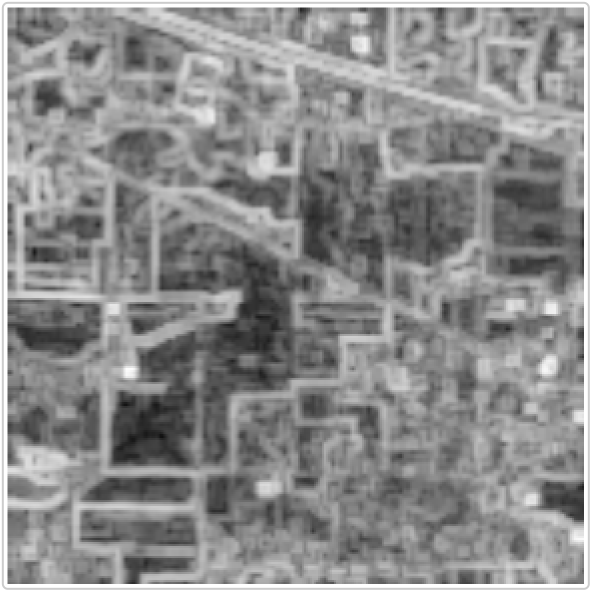
```
summary(ip)
#> The Data name is IndianPines
#> This image is 145 x 145 pixels with 200 layers.
#> The RGB bands are:  44 22 12
#> The gt levels are:
#>  [1] "0 Undefined"                  "1 Alfalfa"
#>  [3] "2 Corn-notill"                "3 Corn-mintill"
#>  [5] "4 Corn"                       "5 Grass-pasture"
#>  [7] "6 Grass-trees"                "7 Grass-pasture-mowed"
#>  [9] "8 Hay-windrowed"              "9 Oats"
#> [11] "10 Soybean-notill"            "11 Soybean-mintill"
#> [13] "12 Soybean-clean"             "13 Wheat"
#> [15] "14 Woods"                     "15 Buildings-Grass-Trees-Drives"
#> [17] "16 Stone-Steel-Towers"
plot(ip)
```

Watershed Segmentation: Gradient

In order to use the Watershed segmentation algorithm, we need to find the gradient for the image. We calculate the Robust Color Morphological Gradient as defined in (Tarabalka *et al.*, 2010). First, for every pixel X_p, find the set of e neighboring pixels $\chi = \{X_p^1, X_p^2, \ldots, X_p^e\}$ where $X_p \in \chi$. Let $\chi^r \subset \chi$, such that the r pairs of pixels with the largest distance have been removed. Second, find the RCMG as $\nabla^{RCM}_{\chi^r,d} = \max_{i,j\in\chi^r}\{d(X_p^i, X_p^j)\}$, where d is an appropriate distance function. The original paper suggests using the Euclidean distance function, however, there are other options that may be better for HSI, such as Cosine Distance, Spectral Angle Mapper, Euclidean Cumulative Sum, Kullbach-Leibler Pseudo Divergence, and others (Deborah, 2016).

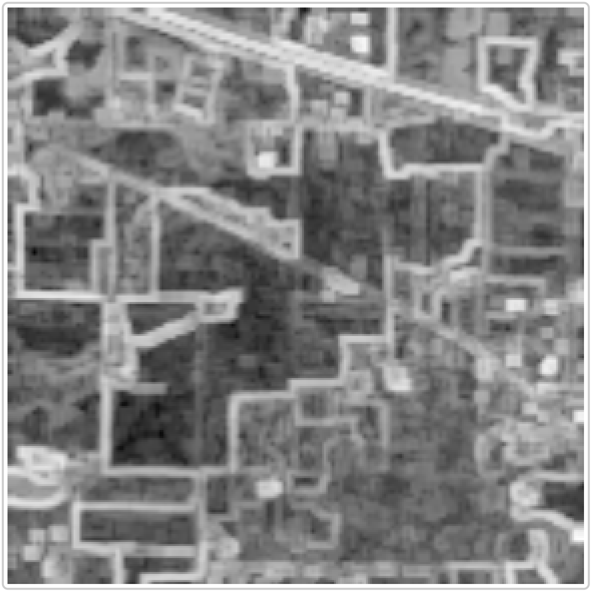
For this package, we implemented a `rcmg_euclidean` and `rcmg_cos` gradient calculation which implements the euclidean and cosine distances respectively. Each is implemented in Rcpp armadillo to help reduce computation time. It is recommend to use `calc_grad` function for error checking, additionally it returns the gradient as an `HSI_grad` allowing one to use the generic `plot` function to visualize the gradient. Note that the gradient plots have darker colors (black) represent low gradient (high similarity between neighboring pixels), while white indicates larger gradient (low similarity between neighboring pixels. . For example, we can calculate the RCMG using Euclidean distance with:

```
grad_e <- calc_grad(ip)
plot(grad_e, log = TRUE)
```



Additionally, we can use the cosine distance function, which produces a slightly different gradient:

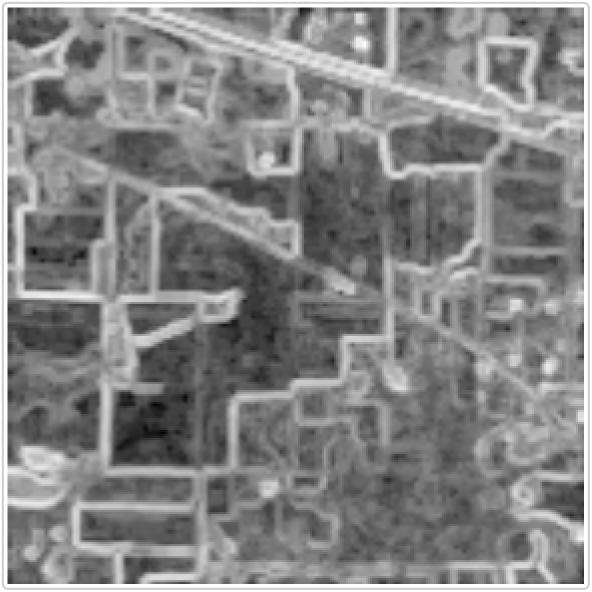
```
grad_c <- calc_grad(ip, "cos")
plot(grad_c, log = TRUE)
```



Another parameter that can be tuned is r, which is the number of pairs of pixels to remove. This will create overall smaller gradient values when you use a larger r. For example, here is the Euclidean RCMG using $r = 2$:

```
grad_2 <- calc_grad(ip, "cos", 2)
```

```r
plot(grad_2, log = TRUE)
```



Note that in all these plots we specify a plotting argument `log=TRUE`. This will take the log of the gradient, which helps one to visually compare gradients when using different functions. Furthermore, there are a few key points to note with these gradients. We see that the gradient appears to do a good job at finding the boarders between distinct fields in the image. Additionally, we see that all the gradients capture very similar information, though the cosine gradient appears to have somewhat less noise compared to the Euclidean gradient. And comparing the two cosine gradients using different `r` values, we see that when using `r=2` the gradient seems a little smoother around the strong edges compared to when `r=1`.

It should be noted that one parameter we don't allow to change is the window size. Our implementation sets the window size to be 1, which corresponds to finding the 8 neighboring pixels. For more gradient options, users should use the Python `SpecAI.Seg` package.

Watershed Segmentation: Segmentation

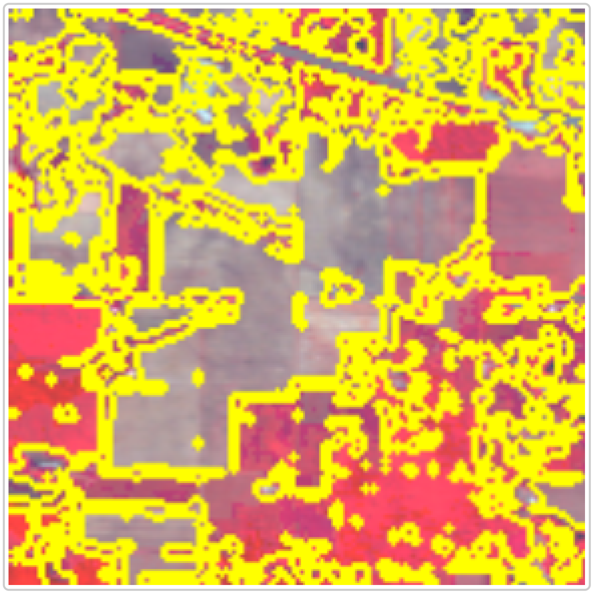
Once the gradient of the HSI has been calculated, `watershed_hsi` allows users to segment that gradient into distinct pieces using the watershed algorithm.

```r
seg <- watershed_hsi(grad_e, tolerance = 1, ext = 200)
```

`watershed_hsi` takes three arguments, a gradient (or object of class `HSI_data`), a tolerance parameter, and an ext parameter. The gradient will typically be what is returned by `calc_grad`. The tolerance parameter is dependent on the scale of your gradient, and will combine objects with a height difference that is less than tolerance. The ext parameter functions as a way to smooth out the segmented image. This implementation of the watershed algorithm also minimizes gradient noise by ignoring the lower quantiles of said gradient. `watershed_hsi` will return an object of class `HSI_seg`, which contains the segmented image, and the gradient used as an attribute of the object.

```r
marked_img <- mark_boundaries(seg, ip$img_rgb, c(1, 1, 0))
```

```r
ggmap::ggimage(marked_img)
```

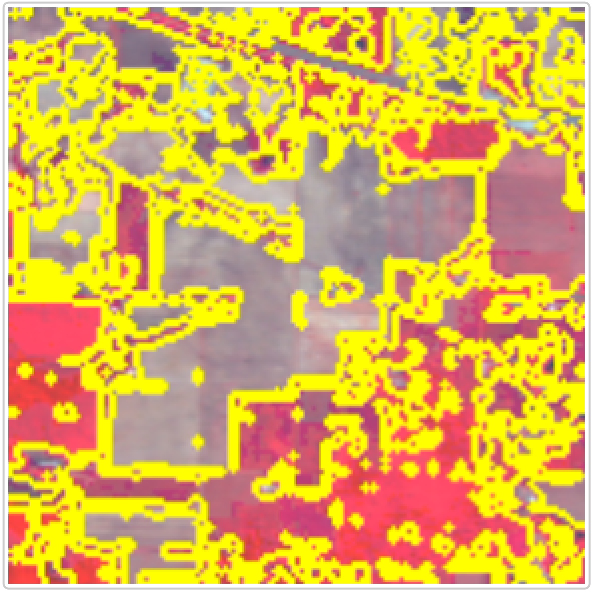


`mark_boundaries` takes the segmented gradient, an RGB version of the image, and an RGB color vector. It uses erosion and dilation to return an “image” (in matrix form) with the boundaries of the segmented image marked on the original image, typically in yellow. These yellow lines represent where our image should be segmented according to the watershed algorithm, i.e. the yellow lines should be separating distinct parts of the original hyperspectral image. In the example above, we see the yellow lines typically tracing the outside of the fields, which is what we would expect. It may also be capturing other behaviors that are hard to see in the original RGB image. The color vector can be changed freely, but does have a default of yellow (maximum red and green). This function will typically be used automatically when plotting an object of class `HSI_seg`.

Plotting and Summaries

The segmentation results can again be easily plotted using plot, this time of `HSI_seg` form, with the segmentation and the dataset as arguments:

```r
plot(seg, ip)
```



Conclusions

This R package brings some of the basic functionality of the original Python `SpecAI.Seg` package to R. Though the full functionality has not been implemented, there is a solid foundation of functions to allow users to start exploring hyperspectral image segmentation in R, starting with the Watershed segmentation algorithm. This package uses simple array representation of hyperspectral images, which helps to simplify the work flow with these images compared to having to use full spatial dataframe objects. Furthermore, this is, to the best of our knowledge, the first R package to allow for hyperspectral image segmentation and visualization.

There is future work to do in order flesh out this package. First is to allow users to load in locally stored images, instead of only being able to download the set 6 hyperspectral images we provide. Second, implement a new Watershed segmentation algorithm that produces results closer to the Python implemented algorithm. Third, implement many more segmentation algorithms, such as Felzenszwalb, Quickshift, and Slic.

References

Deborah H. (2016). Towards spectral mathematical morphology. Université de Poitiers; Norwegian University of science and technology ….

Jarman S. (2022). SpecAI.seg. https://pypi.org/project/SpecAI.Seg/.

Tarabalka Y., Chanussot J. & Benediktsson J.A. (2010). Segmentation and classification of hyperspectral images using watershed transformation. Pattern Recognition 43 (7): 2367–2379.