DBSCAN

May 11, 2023

[ ]:
```
:dep ndarray = "0.15.6"
```

[2]:
```
:dep linfa = { version = "0.6.1" }
// clustering is a separate crate
:dep linfa-clustering = { version = "0.6.1" }
```

[4]:
```
// apt install libfontconfig-dev on Ubuntu 22.04 LTS
:dep plotters = { version = "^0.3.4", default_features = false, features =␣
↪["evcxr", "all_series"] }
extern crate plotters;
// Import all the plotters prelude functions
use plotters::prelude::*;
```

[5]:
```
:dep rand = "0.8.5"
```

[6]:
```
// Import the linfa prelude and KMeans algorithm
use linfa::prelude::*;
use linfa_clustering::Dbscan;
// We'll build our dataset on our own using ndarray and rand
use ndarray::prelude::*;
// Import the plotters crate to create the scatter plot
use plotters::prelude::*;
```

[7]:
```
use rand::prelude::*;
use std::f32;
```

[8]:
```
// Creates random points contained in an approximately square area
pub fn create_square(center_point: [f32; 2], edge_length: f32, num_points:␣
↪usize) -> Array2<f32> {
    // We
    let mut data: Array2<f32> = Array2::zeros((num_points, 2));
    let mut rng = rand::thread_rng();
    for i in 0..num_points {
        let x = rng.gen_range(-edge_length * 0.5..edge_length * 0.5); //␣
↪generates a float between 0 and 1
        let y = rng.gen_range(-edge_length * 0.5..edge_length * 0.5);
        data[[i, 0]] = center_point[0] + x;
```

```
        data[[i, 1]] = center_point[1] + y;
    }

    data
}
```

[9]:
```
// Creates a circle of random points
pub fn create_circle(center_point: [f32; 2], radius: f32, num_points: usize) ->␣
 ↪Array2<f32> {
    let mut data: Array2<f32> = Array2::zeros((num_points, 2));
    let mut rng = rand::thread_rng();
    for i in 0..num_points {
        let theta = rng.gen_range(0.0..2.0 * f32::consts::PI);
        let r = rng.gen_range(0.0..radius);
        let x = r * f32::cos(theta);
        let y = r * f32::sin(theta);

        data[[i, 0]] = center_point[0] + x;
        data[[i, 1]] = center_point[1] + y;
    }

    data
}
```

[10]:
```
// Creates a line y = m*x + b with some noise
pub fn create_line(m: f64, b: f64, num_points: usize, min_max: [f64; 2]) ->␣
 ↪Array2<f64> {
    let mut data: Array2<f64> = Array2::zeros((num_points, 2));

    let mut rng = rand::thread_rng();
    for i in 0..num_points {
        let var_y = rng.gen_range(-0.5..0.5f64);
        data[[i, 0]] = rng.gen_range(min_max[0]..min_max[1]);
        data[[i, 1]] = (m * data[[i, 0]]) + b + var_y;
    }

    data
}
```

[11]:
```
// Creates a quadratic y = m*x^2 + b with some noise
pub fn create_curve(m: f64, pow: f64, b: f64, num_points: usize, min_max: [f64;␣
 ↪2]) -> Array2<f64> {
    let mut data: Array2<f64> = Array2::zeros((num_points, 2));

    let mut rng = rand::thread_rng();
    for i in 0..num_points {
        let var_y = rng.gen_range(-0.5..0.5f64);
```

```
        data[[i, 0]] = rng.gen_range(min_max[0]..min_max[1]);
        data[[i, 1]] = (m * data[[i, 0]].powf(pow)) + b + var_y;
    }

    data
}
```

[12]:
```
// Creates a hollow circle of random points with a specified inner and outer diameter
pub fn create_hollow_circle(
    center_point: [f32; 2],
    radius: [f32; 2],
    num_points: usize,
) -> Array2<f32> {
    assert!(radius[0] < radius[1]);
    let mut data: Array2<f32> = Array2::zeros((num_points, 2));
    let mut rng = rand::thread_rng();
    for i in 0..num_points {
        let theta = rng.gen_range(0.0..2.0 * f32::consts::PI);
        let r = rng.gen_range(radius[0]..radius[1]);
        let x = r * f32::cos(theta);
        let y = r * f32::sin(theta);

        data[[i, 0]] = center_point[0] + x;
        data[[i, 1]] = center_point[1] + y;
    }

    data
}
```

[13]:
```
// Check the array has the correct shape for plotting (Two-dimensional, with 2 columns)
pub fn check_array_for_plotting(arr: &Array2<f32>) -> bool {
    if (arr.shape().len() != 2) || (arr.shape()[1] != 2) {
        panic!(
            "Array shape of {:?} is incorrect for 2D plotting!",
            arr.shape()
        );
        // false
    } else {
        true
    }
}
```

[14]:
```
// The goal is to be able to find each of these as distinct, and exclude some of the noise
let circle: Array2<f32> = create_circle([5.0, 5.0], 1.0, 100); // Cluster 0
```

```
let donut_1: Array2<f32> = create_hollow_circle([5.0, 5.0], [2.0, 3.0], 400); //
 ↪ Cluster 1
let donut_2: Array2<f32> = create_hollow_circle([5.0, 5.0], [4.5, 4.75], 1000);␣
 ↪// Cluster 2
let noise: Array2<f32> = create_square([5.0, 5.0], 10.0, 100); // Random noise


let data = ndarray::concatenate(
        Axis(0),
        &[circle.view(), donut_1.view(), donut_2.view(), noise.view()],
    )
    .expect("An error occurred while stacking the dataset");
```

[19]:
```
evcxr_figure((640, 240), |root| {

    root.fill(&WHITE).unwrap();

    let x_lim = 0.0..10.0f32;
    let y_lim = 0.0..10.0f32;

    let mut ctx = ChartBuilder::on(&root)
        .set_label_area_size(LabelAreaPosition::Left, 40) // Put in some margins
        .set_label_area_size(LabelAreaPosition::Right, 40)
        .set_label_area_size(LabelAreaPosition::Bottom, 40)
        .caption("DBSCAN", ("sans-serif", 25)) // Set a caption and font
        .build_cartesian_2d(x_lim, y_lim)
        .expect("Couldn't build our ChartBuilder");

    let circle: Array2<f32> = create_circle([5.0, 5.0], 1.0, 100); // Cluster 0
    let donut_1: Array2<f32> = create_hollow_circle([5.0, 5.0], [2.0, 3.0],␣
 ↪400); // Cluster 1
    let donut_2: Array2<f32> = create_hollow_circle([5.0, 5.0], [4.5, 4.75],␣
 ↪1000); // Cluster 2
    let noise: Array2<f32> = create_square([5.0, 5.0], 10.0, 100); // Random␣
 ↪noise

    let data = ndarray::concatenate(
        Axis(0),
        &[circle.view(), donut_1.view(), donut_2.view(), noise.view()],
    )
    .expect("An error occurred while stacking the dataset");

    // Compared to linfa's KMeans algorithm, the DBSCAN implementation can␣
 ↪operate
    // directly on an ndarray `Array2` data structure, so there's no need to␣
 ↪convert it
```

```
    // into the linfa-native `Dataset` first.
    let min_points = 20;
    let clusters = Dbscan::params(min_points)
        .tolerance(0.6)
        .transform(&data)
        .unwrap();
    // println!("{:#?}", clusters);


    let x_lim = 0.0..10.0f32;
    let y_lim = 0.0..10.0f32;


    ctx.configure_mesh().draw().unwrap();
    let root_area = ctx.plotting_area();
    // ANCHOR_END: configure_chart

    // ANCHOR: run_check_for_plotting;
    // check_array_for_plotting(dataset: &Array2<f32>) -> bool {}
    check_array_for_plotting(&circle); // Panics if that's not true
                                       // ANCHOR_END: run_check_for_plotting

    // ANCHOR: plot_points
    for i in 0..data.shape()[0] {
        let coordinates = data.slice(s![i, 0..2]);

        let point = match clusters[i] {
            Some(0) => Circle::new(
                (coordinates[0], coordinates[1]),
                3,
                ShapeStyle::from(&RED).filled(),
            ),
            Some(1) => Circle::new(
                (coordinates[0], coordinates[1]),
                3,
                ShapeStyle::from(&GREEN).filled(),
            ),
            Some(2) => Circle::new(
                (coordinates[0], coordinates[1]),
                3,
                ShapeStyle::from(&BLUE).filled(),
            ),
            // Making sure our pattern-matching is exhaustive
            // Note that we can define a custom color using RGB
            _ => Circle::new(
                (coordinates[0], coordinates[1]),
                3,
```

```
                ShapeStyle::from(&RGBColor(255, 255, 255)).filled(),
            ),
        };

        root_area
            .draw(&point)
            .expect("An error occurred while drawing the point!");
    }

    Ok(())
}).style("width:90%")
```

[16]:
```
extern crate plotters;
use plotters::prelude::*;
evcxr_figure((640, 240), |root| {
    // The following code will create a chart context
    let mut chart = ChartBuilder::on(&root)
        .caption("Hello Plotters Chart Context!", ("Arial", 20).into_font())
        .build_cartesian_2d(0f32..1f32, 0f32..1f32)?;
    // Then we can draw a series on it!
    chart.draw_series((1..10).map(|x|{
        let x = x as f32/10.0;
        Circle::new((x,x), 5, &RED)
    }))?;
    Ok(())
}).style("width:60%")
```

[ ]:
```

```