F20-21DV Data Visualisation and Analytics

Lab 2 – Data Transformations

Preamble

Like with Lab 1, here are a few words of advice.

Tutorials may seem abstract until you get to practise their content in exercises. This is normal. Hence you might want to read them *at least* twice: before you get to the related exercises, to get some context; and after the exercises, to make sure you understand what you did.

We won't be able to cover everything in detail. However, each section provides links to relevant documentation or examples for you to go further. You are strongly encouraged to read these to (a) get an idea of the scope of what you can do, and (b) to inspire your creativity for the coursework and help you demonstrate your mastery of the topic.

It can be tempting to simply skip most of the text and jump to code snippets and copy paste them in your projects and expect great results …

1. Not reading the tutorials and text accompanying exercises means you don't get the context of these code snippets and skip the part where you learn things.
2. Most of the code snippets are succinct and focus on one thing. You are required to understand where it fits in the overall project, and sometimes, fill in the blanks that were left out purposefully.

Make sure you read through this document in its entirety.

In Lab 1, we looked at creating reusable charts, each relying on data structured in a particular way. However, when you get a dataset, it is rarely in the format you specifically designed for your chart. Your application therefore requires you to both load and transform dataset on the fly, so that it fits your chart specifications.

In this second lab, we will explore:

1. JavaScript data structures: their types, and some native data transformation methods available for each type.
2. D3 methods for transforming data structures and extracting information.
3. D3 methods for parsing and formatting data.
4. D3 methods for loading datasets from file.

By the end of the lab, you should be able to pick and use a combination of these methods in order to build data subset for your charts.

Tutorial 5 – D3 Fetch

The first thing we will talk about is: *how to load data from a file into your application?*

In pure JavaScript, the Fetch API provides a standard set of operations for fetching (querying and receiving) resources of any type, and loading these resources into your web applications.

Datasets can come in a wide variety of formats, but typically we tend to use:

- Plain text (*.txt*)
- Comma Separated Values, or CSV (*.csv*)
- JavaScript Object Notation, or JSON (*.json*)
- EXtensible Markup Language, or XML (*.xml*)

Each have their own advantages and limitations.

CSV is an efficient way to store tabular data with fixed headers. JSON and XML can be more hierarchical, with JSON offering greater flexibility.

The native *fetch()* method typically returns an object of type *response*, that you are then meant to parse. Because we tend to parse CSV and JSON a lot, D3 provides a module dedicated to methods that build on top of the Fetch API: **D3 Fetch**.

You can check the documentation there: d3js.org/d3-fetch.

Hence to fetch CSV or JSON data from a URL, you use the methods:

`d3.csv(`*URL*`)`

`d3.json(`*URL*`)`

Now, because these data loading can take time, and JavaScript is a *just-in-time* compiled language, it means that your program is likely to move on to the next instruction **before** your data is fully loaded. These two operations above therefore return a *Promise* object, not the actual data.

To get the data, you must instruct your program to explicitly wait for the operations to complete:

```
let data = await d3.csv(URL);
//or
let data = await d3.json(URL);
```

JSON data is typically already formatted to encode different data types. CSV, however, treats every attribute as a String type. With the **`d3.csv`** operation, we can provide an extra parameter to parse each row:

```
let data = await d3.csv("file.csv", (d) => {
    // for each row d, return a new object
    return {
        year: new Date(+d.Year, 0, 1), // convert "Year" column to Date
        make: d.Make,
        model: d.Model,
        length: +d.Length, // convert "Length" column to number
        registered: d.Reg==='Yes', // convert "Reg" column to boolean
```

```
        large: +d.Length > 100 // create new boolean attribute
    };
});
```

Exercise – Loading Data

You can keep working on the same project as in Lab 1. The exercises in this lab will focus on applying data transformations, and checking the results of such transformations in the Web Console.

You can either work in your existing **`main.js`** script, or create a new one (e.g., **`main2.js`**). Make sure that the script you are working on is loaded by your **`index.html`** page.

Along with this Lab document, you will find a CSV file on Canvas. The file contains a mock dataset of movies. For each row, the following attributes are recorded:

- release_year – the movie's year of release (2010 to 2020)
- release_month – the movie's month of release (1 to 12)
- genre – the movie's genre, e.g., *Comedy*, *Drama*, *Action*, etc.
- director – the movie's fictional director name, e.g., *Miss Scarlett*, *Colonel Mustard*, etc.
- budget – the movie's budget, a float
- revenues – the movie's revenues, a float
- ratings_A – the movie's rating from a fictional website, a float between 0 and 10
- ratings_B – the movie's rating from another fictional website, an integer between 1 and 5
- ratings_C – the movie's rating from a third fictional website, an integer between 0 and 100

Download this CSV file and save in your project's directory, for example in a new subdirectory called ***`data`***.

Using the code above as example, your task is to load this dataset into your application. For now you do not need to parse rows. Using the **`console.log`** operation, print this dataset in the Web Console.

How many records are there in the dataset?

In what format is the dataset structured, inside the application?

With this information in mind, write a row parser to correctly format the dataset. Once you have correctly parsed the existing attributes, improve your parser to add two extra attributes: the profits for each movie (revenue-budget), and whether it was a commercial success (positive profit).

Tutorial 6 – JS Data Structures and Operations

Now let's dive into JavaScript's Data Structures their native operations (i.e., not provided by D3).

As a quick revision, a *Data Structure* is a programming language's way to store, organise and provide access to a collection of data.

Data Structures

Since ECMAScript 6/2015, JavaScript implements 4 main types of data structures.

- **Object** – a collection of values identified with a string key
- **Array** – a collection of values identified with an integer index
- **Map** – a collection of values identified with a key of any type
- **Sets** – a collection of unique values

Objects and Maps may seem redundant (Objects were the old hack way to implement Maps before they were fully introduced). But there are differences:

- Objects have a prototype, with pre-existing keys and values
- Map keys can be of any type, but Object keys must be strings
- Maps have an order (by keys) and are therefore iterable, Objects aren't.
- JSON supports Objects but not Maps

To declare these data structures:

```
let myObject = {}; // or
let myObject = new Object();

let myArray = []; // or
let myArray = new Array();

let myMap = new Map();

let mySet = new Set();
```

You can also easily convert between iterables:

```
let myMap = new Map([[1,'one'],[2,'two'],[1,'three']]);

let mySet = new Set([1,2,3,4,1,3]);

let myArray1 = Array.from(mySet);

let myArray2 = Array.from(myMap);
```

Data Operations

Each of these data structures has plenty of operations that you can use to transform datasets and create subsets for your visualisations.

We can cover all of these methods, so make sure to read through their documentation to get a more exhaustive view:

- Object – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
- Array – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- Map – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
- Set – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Object

There are usually 4 interesting methods you can use with objects. These methods are actually static, meaning that you must invoke them with the object class name, for instance:

```
let myObject = {'A':1, 'B':2};
let keys = Object.keys(myObject); // ['A', 'B']
```

`Object.keys(obj)` returns an array of all keys in the object.

`Object.values(obj)` returns an array of all values in the object.

`Object.entries(obj)` returns an array of all key/value pairs in the object.

Finally, **`Object.groupBy(`*`array, callback`*`)`** applies the callback on each element of the given array. This callback must return a string value. Then, using the string returned, the operation will return a new object with nested arrays, grouped by the string value attached to each entry. Check the great examples on the MDN documentation website to get a better idea: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/groupBy#using_object.groupby. You should note that this last operation is not yet supported on browsers like Safari (Chrome and Firefox are fine though).

Array

Array has a lot more operations. We will highlight important ones here, but again, you are **strongly** invited to check the link above for more details.

First, the length *property* will give the number of items in the array.

```
let myArray = [9,10,3,5,6];
myArray.length // 5
```

Then, contrary to objects, there are a lot more instance methods, meaning that you invoke them using the variable name, for instance:

```
let myArray = [9,5,6];
myArray.push(4); // myArray -> [9,5,6,4]
```

A final remark, when using an Array operation, you should check if this operation is *mutating* the array or *copying* the array.

- Mutating methods will change the array from which you invoke the operation.
- Copying methods will make a shallow copy of the array, and then modify and return the copy. The original array remains unchanged.

Mutating operations:

- **`array.push(value)`** – adds a new element at the end of the array
- **`array.pop()`** – removes (and returns) the last element in the array
- **`array.shift(value)`** – adds a new element at the start of the array
- **`array.unshift()`** – removes and returns the first element in the array
- **`array.concat(otherArray)`** – concatenates the given array to the existing one
- **`array.sort(comparator)`** – sorts the array given a comparator function
- **`array.reverse()`** – reverses the order of the array

Copying operations:

- **`array.map(function)`** – returns a new array where the given function has been applied to each element of the original array
- **`array.filter(test)`** – returns a new array where only the elements for which the given test function returned true
- **`array.flat(depth)`** – returns a new flatten array, up to the depth provided
- **`array.slice(start, end)`** – returns a new sliced array with only the element in the range corresponding to the start and end (excluded) indices
- **`array.toSorted(comparator)`** – returns a new sorted array, using the given comparator function
- **`array.toReversed()`** – returns a new reversed array

Querying operations (return information about the array):

- **`array.forEach(callback)`** – executes the callback for each element in the array
- **`array.every(test)`** – checks if the test function holds true for all elements in the array
- **`array.some(test)`** – checks if the test function holds true for at least one element in the array
- **`array.includes(value)`** – checks if the array contains the given the value
- **`array.reduce(function,value)`** – reduces the array to a single value, given the accumulator function and the initial value provided
- **`array.join(string)`** – returns a string with all elements, joined by the given string

Map and Filter are extremely powerful operations when it comes to transforming datasets. Reduce is also a popular one, for generating aggregated values.

For instance, imagine this set of data:

```
let data = [{name:'Pete',age:26},{name:'Johanna',age:27},
    {name:'Jamie',age:38},{name:'Aleksy',age:29},
    {name:'Akira',age:25},{name:'Adesina',age:32},
    {name:'Lorenzo',age:30},{name:'Julien',age:26}];
```

Now, let's say we wanted to get the average name length of those between the ages of 26 and 34, and with a name starting with the letter J:

```
let sizes = data.filter(d=>d.age<35 && d.age>25) // filter ages
    .map(d=>d.name) // grab names only
    .filter(d=>d.startsWith('J')) // filter first letters
    .map(d=>d.length); // map to name length
```

```
let avgSize = sizes.reduce((a,b)=>a+b, 0)/sizes.length; // sum sizes
divided by length
```

Map

Map (and Set) where introduced a bit later in the history of JavaScript. As the language grew in maturity, you will find that the operations for these data structures are a defined in a better way.

- **`map.size`** – returns the number of entries in the map
- **`map.set(key, value)`** – sets the value for the given key
- **`map.get(key)`** – returns the value for the given key
- **`map.has(key)`** – checks that the map contains an entry for the given key
- **`map.delete(key)`** – deletes the entry associated with the given key
- **`map.clear()`** – deletes all entries
- **`map.keys()`** – returns an array of all keys
- **`map.values()`** – returns an array of all values
- **`map.entries()`** – returns an array of all key-value pairs
- **`map.forEach(callback)`** – executes the given callback on each entry

Set

- **`set.size`** – returns the number of elements in the set
- **`set.add(value)`** – adds the given value
- **`set.has(value)`** – checks if the given value exists
- **`set.delete(value)`** – deletes the given value from the set
- **`set.clear()`** – deletes all items from the set
- **`set.forEach(callback)`** – executes the given callback on each item

On Safari, there also are many experimental methods available for Sets: set difference, set intersection, set union, super/sub set checks.

As a general tip when using all these structures and operations, you should regularly print the result of the operation you are preforming. That way you can verify that they are doing what you think you are doing, and get a clearer view of what you can do next.

Exercise – Basic Transformations

Using some of the structures and operations described above, answer this series of questions about the movie dataset.

- What are the unique genres of movie?
- How many unique directors are there?
- What's the total sum of revenue in the industry?
- How many movies were released between 2012 and 2014 (included)?
- What is the average rating on website A for comedy movies?
- Has the industry made more profit before 2015 (included) or after?
- What's the average budget of drama movies with a rating above 70% on website C?

Tutorial 7 – D3 Array

D3 Array is a module offered by D3 that provides even more functions to explore and refine datasets. These can be used to transform datasets for visualising them, or grab information on the fly to customise scales or (later) interactions.

You can find the documentation at this address: https://d3js.org/d3-array

We can use this module for:

- Getting descriptive statistics
- Search for values
- Transforming datasets, notably by aggregation
- Extend built-in array functions like map, filter and reduce to Maps and Sets
- Perform set operations (union, intersection, etc.)

Rather than list all methods here, you should read the following documentation pages:

- https://d3js.org/d3-array/summarize (**Important**)
- https://d3js.org/d3-array/transform (**Important**)
- https://d3js.org/d3-array/sort (Useful to know)
- https://d3js.org/d3-array/sets (Useful to know)
- https://d3js.org/d3-array/bisect (Useful to know)

Grouping

For years, JavaScript didn't include a means to group data from an array. With recent development (ca. 2023) we are getting there.

In the meantime, D3 soon provided its own methods for grouping data, and in fact it still has a more powerful set of operations for this purpose.

This page can give you more information: https://d3js.org/d3-array/group

In short, there are two important categories of operations: *group* and *rollup*.

A **group** operation will aggregate the items according to one or more key accessor functions.

```
d3.group(array, d=>d.key1, d=>d.key2, …)
```

We can note two differences with JavaScript's *groupBy* method: (a) you can provide more than one key accessor, and (b) your key doesn't have to be a string.

If you provide more than one key accessor, you will get a nested grouping:

```
value1 for key1
    value1 for key2
        [subset of entries]
    value2 for key2
        [subset of entries]
value2 for key1
    . . .
```

A **rollup** operation is somewhat similar, in that it will aggregate your data, possibly in nested groups if you have several keys; but it will conclude the aggregation process with a reduce

operation on each of the subset created. For instance, to get the number of item in each subset:

```
d3.rollup(array, D=>D.length, d=>d.key1, d=>d.key2, …)
```

```
value1 for key1
    value1 for key2
        reduced value for subset
    value2 for key2
        reduced value for subset
value2 for key1
    . . .
```

With rollup, the second parameter, the reduce operation, can be anything, a count, a sum, an average, etc…

Group and Rollup come in three variants each:

- d3.group – returns the nested groups as Maps
- d3.groups – returns the nested groups as Arrays
- d3.flatGroup – returns the groups as a flat Array
- d3.rollup – returns the nested reduced groups as Maps
- d3.rollups – returns the nested reduced groups as Arrays
- d3.flatRollup – returns the reduced groups as a flat Array

You should experiment with each of these and check the results in the Web Console to fully understand what you can do with those operations.

Binning

Group and Rollup are concerned with aggregation by discrete values.

When aggregating over a continuous interval, you might be more interested in Binning: https://d3js.org/d3-array/bin

It functions much like the data generators we saw in the previous lab: you first specify a bin generator; you then apply that generator on a dataset.

When setting the bin generator, you can specify:

- A value accessor for the dataset you will provide. Defaults to the identify function.
- The target domain, values outside this domain will be ignored. Defaults to the extent of values.
- The threshold for the bins. You may provide thresholds explicitly, or simply pass a target number of bins (the generator will then compute equally sized bins).

Exercise – Aggregations

Continuing with our movie datasets, and its analysis/transformation, can you use the D3 methods shown above to answer the following questions:

- Group the movies by Director and then by Genre.
- Group the movies by Year and then Genre, and get the number of movies for each subset.

- Distribute the entries into 10 equally-sized categories based on budget values.
- What are the average profits by Director?
- What are the total revenues by Genre?
- Construct a new array, each entry with two values: the Director name and their ratio of commercial success (profitable / total number of movies)
- Are there any common entries in both the top 10 Comedy (by revenue) and the top 10 directed by Professor Plum (by revenue)?

Exercise – Let's Make a Histogram

Using the BarChart class as a starting point, create a class for Histograms.

The Histogram should accept an array of value as dataset input, along with a target count for the number of bins it produces.

It should display bins as bars along a continuous axis, the height of each bar corresponding to the percentage of values in the bin.

You will have to modify the scale used and introduce a dataset transformation inside the class.

Tutorial 8 – D3 Formats

When dealing with data, especially numbers and dates, the format we deal with inside our application can be quite different from the format users can read, or how it's stored in text.

For instance:

- The number `1200345` could be displayed as ***1.2M***.
- The number `0.235` could be shown as ***23.5%***.
- A date could be stored as the string "`03/02/2023`", but internally considered as `2023-02-03T00:00:00.000Z`, and finally displayed as ***Feb. 2023*** to users.

D3 has two modules to deal with these things:

- D3 Format (https://d3js.org/d3-format) for numbers; and
- D3 Time Format (https://d3js.org/d3-time-format) for dates/times.

D3 Format

D3 Format has two purposes: consistency and interpretability of numbers.

Representing numbers, especially floating-point ones, is actually a challenging topic for programming languages. For that reason, sometimes programs can run into problems when calculating decimals and you mysteriously end up with a value like `3.00000001`, when you really were expecting something more like `3`.

On a similar note, when we decide to print number data to users, we have conventions to summarise very long numbers (either really big or really small) efficiently. For example, `50603045.67` can be written "***50,603,045.67***", or rounded to take magnitude into account, like "***5.06e+7***" or "***50.6M***".

To achieve this, it let's you generate a formatter function. Given a *specifier*, this formatter will read numbers you provide it and output a string with a consistent format.

There are many options for setting up the specifier. You can find a list on the official D3 documentation (link above). There's also a public Observable notebook that lets you explore and tinker with different options: https://observablehq.com/@d3/d3-format.

Here are some notable settings:

- The *sign* (e.g., "+", "-", " " (space)) lets you set how the number's sign should be displayed.
- The *symbol* (e.g., "$") will prepend or append symbols for currency or specific notations like binary or octal.
- The *grouping* (",") will add group separators, like commas for thousands.
- The *precision* (e.g., ".2" or ".3") sets the number of significant digits or decimal points (depending on the notation you choose).
- The *trailing zeros* option ("~") will remove insignificant trailing zeros.
- Finally, the *notation type* will round the number in different way:
    - "e" for exponential notation
    - "f" for fixed point notation
    - "r" for decimal notation, rounded to the significant digit
    - "d" for decimal notation, rounded to the integer

- “s” for decimal notation, with an SI prefix
- “%” for percentage notation (multiply by 100 and add “%”)

To use formats:

```
let formatter = d3.format(specifier);
let formatNumber = formatter(number);
```

Note that it is also possible to set locale settings too. For instance, French and English numbers are not grouped the same way (French: “1 234,56“, English: “1,234.56”), the currency symbol will be different across countries too.

D3 Time Format

D3 Time Format unique purpose is only to translate between human readable dates and times (“***12th January 2024 at 1:34pm***”) to ISO 8601 formats (“***2024-01-12T13:34:00.000Z***”) which are used by programming languages to work with the (very) human notion that is time.

It provides two sets of function: *parsers* to read strings and generate Date objects; and *formatters* to print Date objects as strings.

Like number formatters, both parsers and formatters are functions that are first generated using a *specifier* string, with plenty of directives to parse/format date-time strings. You can find the complete list of directives here: https://d3js.org/d3-time-format#locale_format.

For instance:

```
let parser = d3.utcParse('%d %B %Y');
let dateObject = parser('18 April 2019');
console.log(dateObject); // "2019-04-18T00:00:00.000Z"

let formatter = d3.utcFormat('%b. %y');
let dateString = formatter(dateObject);
console.log(dateString); // "Apr. 19"
```

Because time scales use ISO 8601 Date objects for their domain, time parsers become incredibly useful if you want to create time-series visualisations. (You can review the line chart solution provided for Lab 1 to get a usage example).

D3 Formats and Axes

When using D3 axes, the **`.tick`** and **`.tickFormat`** methods of your axis generators allow you to set the format (number or date) in which tick marks are printed. You need to provide the appropriate specifier as the methods’ argument.

Exercise – Formatting Data

Using JavaScript data transformations, D3 scales, D3 aggregations and D3 formats, reshape the movie dataset such that:

- There is a single release attribute in the format: “<long-month> <4 digit year>”.
- There is an average rating (out of all 3) in the percentage format.
- Entries are sorted by profit.
- There’s only the profit attribute (no budget or revenues), in the SI prefixed format with 4 significant digits.

- Entries are nested by movie genre and then directors

There are many things to do here, and the order of operation will be important. Don’t try to solve everything all at once. Just build an incremental solution, with regular console prints to verify what you are doing works as expected.