Analysis of a dataset using data mining techniques.

This notebook focuses on exploring and mining basic information from an anonymised retail transactions dataset given by the company Instacart.

Taking a first glance at the dataset

Source code for this section may be find in file `dist/first-glance.class.ts`

The dataset consists of information about 3.4 million grocery orders, distributed across 6 `.csv` files listed below:

In [1]:
```
import * as child_process from 'child_process';

/**
 * Function executes a child_process listing the files in the instacart_basket_data folder
 * and returns the listed files though a Promise.
 */
function listFiles(): Promise<string[]> {
    // Async behavior
    return new Promise( (resolve,reject) => {
        // Listing the files in the instacart_basket_data folder.
        let ls: child_process.ChildProcess = child_process.exec('ls instacart_basket_data', (err: Error, stdout: string, stderr: string) => {
            // Formatting ls output as an Array of strings (representing the file names)
            resolve(stdout.match(/[^\r\n]+/g));
        });
    });
}

function showFiles(): void {
    // Async console for Jupyter
    $$async$$ = true;

    listFiles().then( (files: string[]) => {
        console.log(files);
        $$done$$();
    }, (e) => console.log(e) );
}

showFiles();
```

```
[ 'aisles.csv',
  'departments.csv',
  'order_products__prior.csv',
  'order_products__train.csv',
  'orders.csv',
  'products.csv' ]
```

Files composing the dataset are listed below:

```
[ 'aisles.csv',
  'departments.csv',
  'order_products__prior.csv',
  'order_products__train.csv',
  'orders.csv',
  'products.csv' ]
```

As a starting point, in order to have a first glance of the data we will actually be playing with throughout this entire report; let us display the first items composing each `.csv` file listed above.

We will use NodeJS's straight-forward File I/O fs to list, open and `'csv-parse'`'s `Parser` to parse `.csv` files.

In [3]:

```
import * as fs from 'fs';
import { Parser } from 'csv-parse';

/**
 * Function reads a .csv file and returns it properlly formatted.
 */
function readFile<T>(filePath: string): Promise<{ data:Array<T>, file: string }> {
    // Async console for Jupyter
    $$async$$ = true;
    // Async behavior
    return new Promise( (resolve, reject) => {
        let ret: Array<T> = [];

        // 'csv-parse' Parser, columns options groups each row by column in an object
        let parser: Parser = new Parser({
                delimiter: ',',
                columns: true
        });

        parser
            .on('data', (data: T) => ret.push(data) )
            .on('error', (error: any) => reject(error) )
            .on('end', () => resolve({data: ret, file: filePath}) );

        // Reading the file and piping to parser
        fs.createReadStream(filePath).pipe(parser);
    });
}

listFiles().then( (files: string[]) => {
    console.log('Parsing all the files, this may a while...');
    // Creating promises array to run readFile() of files concurrently
    let promises: Promise<any>[] = files.map( (file: string) => {
        // Formatting to get absolute paths to files.
        let filePath: string = `custom_data/small_${file}`;
        // Reading file
        return readFile<any>(filePath);
    });
    // Concurrent execution of promises
    return Promise.all(promises);
}).then( (results: Array<any[]>) => {
    // Reading first two elements of each file
    results.forEach( (result: Array<any>) => {
        console.log(`First two lines of file: ${result.file}`);
        console.log(result.data.slice(0,2));
    });
    // Async console for Jupyter
    $$done$$();
}).catch( (e) => console.log(e) );
```

```
Parsing all the files, this may a while...
First two lines of file: custom_data/small_aisles.csv
[ { aisle_id: '1', aisle: 'prepared soups salads' },
  { aisle_id: '2', aisle: 'specialty cheeses' } ]
First two lines of file: custom_data/small_departments.csv
[ { department_id: '1', department: 'frozen' },
  { department_id: '2', department: 'other' } ]
First two lines of file: custom_data/small_order_products__prior.csv
[ { order_id: '2',
    product_id: '33120',
    add_to_cart_order: '1',
    reordered: '1' },
  { order_id: '2',
    product_id: '28985',
    add_to_cart_order: '2',
    reordered: '1' } ]
First two lines of file: custom_data/small_order_products__train.csv
[ { order_id: '1',
    product_id: '1',
    add_to_cart_order: '1',
    reordered: '1' },
  { order_id: '2',
    product_id: '101',
    add_to_cart_order: '2',
    reordered: '1' } ]
First two lines of file: custom_data/small_orders.csv
[ { order_id: '2539329',
    user_id: '1',
    eval_set: 'prior',
    order_number: '1',
    order_dow: '2',
    order_hour_of_day: '08',
    days_since_prior_order: '' },
  { order_id: '2398795',
    user_id: '1',
    eval_set: 'prior',
    order_number: '2',
    order_dow: '3',
    order_hour_of_day: '07',
    days_since_prior_order: '15.0' } ]
First two lines of file: custom_data/small_products.csv
[ { product_id: '1',
    product_name: 'Chocolate Sandwich Cookies',
    aisle_id: '61',
    department_id: '19' },
  { product_id: '2',
    product_name: 'All-Seasons Salt',
    aisle_id: '104',
    department_id: '13' } ]
```

Out[3]: undefined

File `aisles.csv` is structured as such; and contains the two following rows:

aisle_id	aisle
1	prepared soups salads
2	specialty cheeses

File `departments.csv` is structured as such; and contains the two following rows:

department_id	department
1	frozen
2	other

File `products.csv` is structured as such; and contains the two following rows:

product_id	product_name	aisle_id	department_id
1	Chocolate Sandwich Cookies	61	19
2	All-Seasons Salt	104	13

Files `aisles.csv`, `departments.csv`, `products.csv` and all (trivially) contain information about the aisles, products, and departments names respectivelly; which may not be of any interet to us other than for enhanced visualisation. We'll thus have to perform the proper `JOIN`s (a.k.a. `UNION`) between these tables and our future data / pattern collections when needed.

File `orders.csv` is structured as such; and contains the two following rows:

order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_order
2539329	1	prior	1	2	08	
2398795	1	prior	2	3	07	15.0

File `orders.csv` is surely a bit more interresting, especially having in mind **sequential pattern mining**, as it lists all the orders, and contains information on **when** and **by who** it has been placed.

Finally, files `order_products__train.csv` and `order_products__prior.csv` contain the same, and most valuable information in regard to pattern mining, as they contain products ordered within each order: We will generate our transactions from this file, grouping `product_ids` by `order_id`.

These files are structured as below:

order_id	product_id	add_to_cart_order	reordered
1	49302	1	1
1	11109	2	1

Warming up: First statistics over the dataset

Main perk of using TypeScript could be the language's proximity to the DOM, for data visualisation. Upon this point, will be using `D3.js` library, as well as `zingchart` the visualise our data.

`D3.js` and `zingchart` are libraries allowing us to create an interactive charts with smooth transitions and interaction. If you're reading this of a `.pdf` version, please consider loading this notebook in a browser (application is dockerized), as many graph only make sense when being interacted with (as Chord Diagrams for frequent itemset presentation for instance).

"Front-end" dependencies will be loaded below.

In [3]:
```
function loadDependencies(): void {
    $$.html(`
        <script>
            // Zingchart Library
            $('head').append('<script src="./node_modules/zingchart/client/zingchart.min.js">');
        </script>
    `);
}

loadDependencies();
```

Out[3]:

We'll keep our first analysis of the data simple, and start by computing some simple statistics over the dataset, on the number of orders, products, products per order, etc. In addition to understanding more about the data, this will also allow us to find some appropriate and revelant criteria on which we could base (and reduce) our dataset for extended itemset analysis; as well as giving us a first glance at pattern dedundancy.

Keeping in mind that each row of the `order_products__prior.csv` file is structured as stated above; an interesting approach would be to regroup these objects by both `order_id` and `product_id`, as it would allow us to have a glance over the product distribution through the orders on the first hand, as well as idea of each product's popularity on the other.

To do so, we need to transform the dataset consequently. Thus, let's start by defining `ProductOrder` as the structure of the data outputted by the `.csv` file parser. We will do the same for each data structure of each file composing our dataset:

```
/**
 * Data structure we gather from CSV File.
 */
interface ProductOrder {
    order_id: string,
    product_id: string,
    add_to_cart_order: string,
    reordered: string
}
```

From this point, as we focus on exploiting data from file `order_products__prior.csv`, which contains more than 1 million records; and for the sake of memory usage, we'll try to work on data streams as much as possible, rather than parsing 1-million-elements-cached `Arrays` when it comes to data transformation. We'll thus be using Reactive Programming library `RxJS` in that intent.

Reactive programming is nothing new as it only consists in programming with asynchronous data streams, which languages like JS are basically all about. `RxJS` yet provides us with an amazing and complete approach -as well as a great toolbox of functions- to combine, create and filter such streams easily.

Grouping by orders

Let us define a function allowing us to group `ProductOrder` objects by any key of the `ProductOrder` interface. We'll use in that intent `RxJS`'s `groupBy` method, which basically groups the items emitted by an `Observable` (a.k.a. stream; in our case, the `ReadStream` of the considered file) according to a specified criterion (in our case, either the `product_id` or the `order_id`), and emits these grouped items as `GroupedObservable`s, one `GroupedObservable` per group.

Let's start by defining a `Group<T>` as an object containg an `id` (the grouping criterion basically), as well as an `Array` of whatever item of type `T` we're grouping. This will be one of many product of our function:

```
interface Group<T> {
    id: string,
    items: Array<T>
}
```

We'll also forge ahead (keeping pattern mining in mind) by allowing this method to `.map()` the `ProductOrder` objects to whatever we want (its `id`, `product_name` ...) depending on our need.

Such a function is given below:

In [4]:

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/finally';
import 'rxjs/add/operator/groupBy';
import * as RxNode from 'rx-node';

/**
 * Function returns an Observable of `ProductOrder` group by a defined criterion. You may map the parsed `ProductOrder` to whatever value you want.
 */
function _readAndGroupBy<T>( key: keyof ProductOrder, map: (val: ProductOrder) => T ): Rx.Observable<Group<T>> {
    /**
     * 'csv-parse' Parser, columns options groups each row by column in an object.
     */
    let parser: Parser = new Parser({
        delimiter: ',',
        columns: true
    });

    // Turning native stream into Observable
    return RxNode.fromStream( fs.createReadStream(`instacart_basket_data/order_products__train.csv`).pipe(parser) )
        // Grouping objects by order
        .groupBy( (data: ProductOrder) => data[key] )
        // At this point, we basically have an Observable by group. Thus we need to flatten that.
        .flatMap( (group: Rx.GroupedObservable<string, ProductOrder>) => {
            return group
                // Formatting the data
                .map(map)
                // And flattening the Observable array.
                .reduce( (concat: Group<T>, current: T) => {
                    concat.items.push(current);
                    return concat;
                }, {
                    id: group.key,
                    items: []
                })
        });
}
```

Out[4]: {}

Let us group the `ProductOrder` by their `order_id`. `ProductOrders` will be caracterized by their `product_id` (We'll thus trivially have an list of Orders (or itemsets), as `Arrays` of `product_id`s).

Function above will return us with all the processed groups. We'll compute some basic statistics on these from there, such as:

- The number of orders (number of groups);
- The minimum number of product in an order (minimum of arrays length);
- The maximum number of product in an order (maximum of arrays length);
- The average product number per order (average array length);
- The number of records in the `order_products__prior.csv` file (sum of arrays length);

In [5]:

```
function statsOnOrders(): void {
    $$async$$ = true;
    console.log('Gathering data, this might take a while...');

    /**
     * All the groups.
     */
    let groups: Array<Group<string>> = [];

    /**
     * Product per order for Chart
     */
    let productsPerOrder: Array<number> = [];

    let stats: any = {
        max: 0,
        min: Infinity,
        sum: 0
    }

    /**
     * Reads the file and groups `ProductOrders as intended`
     */

    _readAndGroupBy<string>('order_id', (productOrder: ProductOrder) => productOrder.product_id )
        // Once all groups are loaded, displaying them.
        .finally( () => {
            console.log(`Maximum number of ProductOrders: ${stats.max}`);
            console.log(`Minimum number of ProductOrders: ${stats.min}`);
            console.log(`Average number of ProductOrders: ${stats.sum / groups.length}`);
            console.log(`Total number of ProductOrders: ${stats.sum}`);
            console.log(`Number of itemsets: ${groups.length}`);

            let chartOptions: string = JSON.stringify({
                id: 'productPerOrderChart',
                data: {
                    type: 'bar',
                    title: {
                        text: 'Product per order sorted ascendingly'
                    },
                    scaleY: {
                        label: {
                          text: "Number of products"
                        },
                        item: {
                          fontSize: 10
                        }
                    },
                    scaleX: {
                        label: {
                          text: "Order id"
                        },
                        item: {
                          fontSize: 10
                        }
                    },
                    series: [{ values: productsPerOrder.sort( (a: number, b: number) => a - b) }]
                }
            });

            $$.html(`
                <div id="productPerOrderChart"></div>
                <script>
                  zingchart.render(${chartOptions});
                </script>
            `);

            $$done$$();
        })
        // Note that this behaviour (induced by the flatMap of readAndGroupBy) makes everything pretty much blocking again.
        .subscribe( (group: Group<string>) => {
            // Computing some basic stats on the fly
            stats.max = Math.max(group.items.length, stats.max);
            stats.min = Math.min(group.items.length, stats.min);
            stats.sum += group.items.length;

            // Pushing group to groups.
            groups.push(group);

            // Chart display data
            productsPerOrder.push(group.items.length);
        });
}

statsOnOrders();
```

```
Gathering data, this might take a while...
Maximum number of ProductOrders: 80
Minimum number of ProductOrders: 1
Average number of ProductOrders: 10.552759338155157
Total number of ProductOrders: 1384617
Number of itemsets: 131209
```

Out[5]:

Grouping the records by their `order_id` enlights us of the following information:

Number of orders	Minimum product number per order	Maximum product number per order	Average product number per order	Total number of records
131,209	1	80	10.6	1,384,617

Eventhough the maximum number of product per order is 80, the graph above shows that approximatly 85% of all the orders only contains between 1 and 20 orders.

Grouping by product

Creating itemsets (`Groups`) of Orders, based on the `product_id` of `ProductOrders` may also be of interest to us, as it allow us "feel" a product "popularity" by counting the number of orders it appears in. This is a pretty good deal in regards to pattern mining, as:

- a "frequent" product will be more likely to appear in frequent itemsets;
- its number of appearance in the collection is, by definition, the maximum support over the dataset. Considering a product A being the most popular in a dataset such as ours, the itemset `{ A }` will trivially be the absolute, most frequent itemset to be find in the entire dataset;
- if a product is too frequent, it may be of interest to ignore it, as the itemsets to be find may not be revealant enough.

Code is basically the same as before, thus won't be included in the notebook. Sources are however still available and results may be parsed from `custom_data/product_id__order_number.csv` .

Grouping the records by their `product_id` gives us the following results:

Number of products	Minimum order number per product	Maximum order number per product	Average order number per product	Total number of records
391,23	1	18,726	35.39	1,384,617

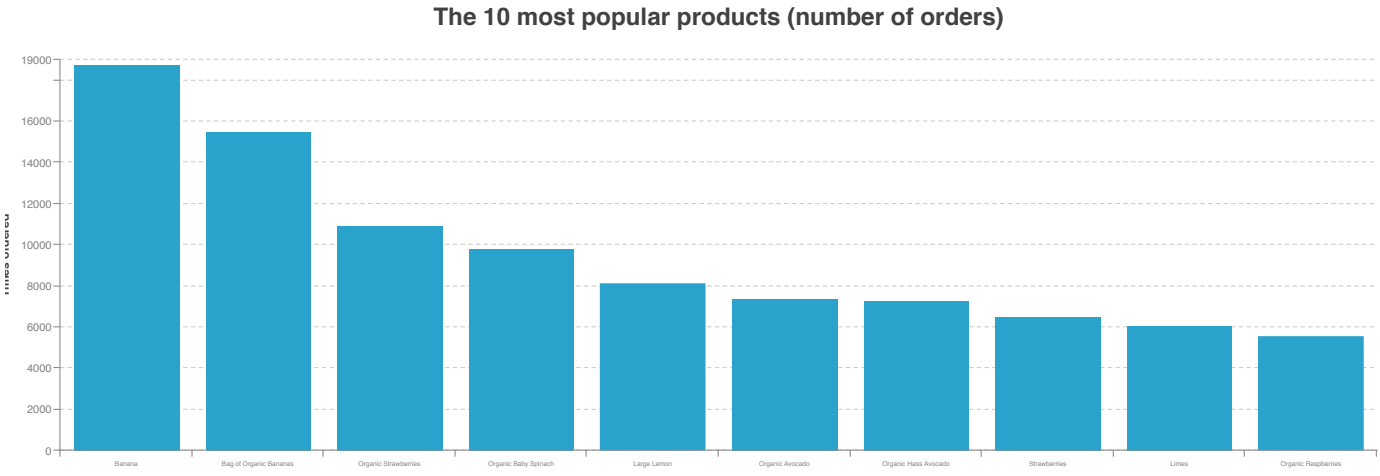
Let us determine the most popular products, by joining the retrieved data with table `products.csv` :

In [6]:

```
import * as CSVParser from './dist/class/csv-parser.class.js';
import * as join from './dist/function/join.function.js'

function joinResultsAndShowPopular(): void {
    $$async$$ = true;
    console.log('Gathering data, this might take a while...');

    // Data to display.
    let chartData: Array<any> = [];

    // Loading the 'product.csv' table.
    new CSVParser.CSVParser<Product>(`instacart_basket_data/products.csv`).loadAll()
        .then( (products: Product[]) => {
            console.log('Finished loading products');

            // Now generating group of items based on the product_id
            new CSVParser.CSVParser<ProductOrder>(`instacart_basket_data/order_products__train.csv`)
                // Grouping items by product_id, and mapping every item composing these itemsets to their order_id.
                .generateItemsets<string>('product_id', (productOrder: ProductOrder) => productOrder.order_id )
                // Once execution is complete, creating chart.
                .finally( () => {
                    console.log('Most popular products :');

                    chartData = chartData
                        // Sorting desc.
                        .sort( (a: any, b: any) => b.number - a.number )
                        // Keeping 10 best products
                        .splice(0,10);

                    let chartOptions: string = JSON.stringify({
                        id: 'popularProductsChart',
                        data: {
                            type: 'bar',
                            title: {
                                text: 'The 10 most popular products (number of orders)'
                            },
                            scaleY: {
                                label: {
                                  text: 'Times ordered'
                                },
                                item: {
                                  fontSize: 10
                                }
                            },
                            scaleX:{
                                values: chartData.map( (product: any) => product.name ),
                                item: {
                                    'font-size':'6px'
                                }
                            },
                            series: [{ values: chartData.map( (product: any) => product.number ) }]
                        }
                    });

                    $$.html(`
                        <div id="popularProductsChart"></div>
                        <script>
                          zingchart.render(${chartOptions});
                        </script>
                    `);

                    $$done$$();

                })
                // Once a group is ready, pushing chart data to the proper array.
                .subscribe( (group: Group<ProductOrder,string>) => {
                    chartData.push({
                        name: join.join<Product>(products, 'product_id', group.id, 'product_name'),
                        number: group.items.length
                    });
                })
        }, (e) => {});
}

joinResultsAndShowPopular();
```

```
Gathering data, this might take a while...
Finished loading products
Most popular products :
```

Out[6]:



Graph above show the 10 most popular products, based on the number of time they've been ordered; **Banana** being the most popular product, with 18,726 orders. From a pattern miner point of view, these products have more chance to be part of multiples itemsets, being that all singleton-itemsets of each of these products are the most frequents.

We can also see that many popular items fall into the same category of product (`Banana & Bag of Organic Bananas / Organic Avocado & Organic Hass Avocado`). If we had some time, regrouping these products would have been a good idea, as more interesting patterns may appear: Nobody would ever buy both Strawberries and Organic Strawberries in the same transaction, yet itemsets `<{Banana , Strawberries}>` and `<{Bag of Organic Bananas , Rasberries}>` **could** be considered as one (if you don't have interest in your consumer `Organic` habits).

Grouping products per aisle (and exploring aisle-to-aisle itemsets) may be a less-time consuming alternative if the need of reducing the dataset becomes apparent: Results would appear as "Someone buying fruits have more chance buying vegetables"; though we obviously loose information on the products bought.

In the meantime, It is worth noticing that **46 products** including `100% Black Cherry & Concord Grape Juice` , `Breaded Popcorn Turkey Dogs` or `Lip Balm` , are the least popular with only 1 order. Filtering less popular products may also be a solution to reducing the dataset; yet I guess this would not change a lot as these elements would still pruned pretty fast by data-mining algorithms when mining for patterns.

Frequent item sets

Knowing a little bit more about our data, we'll now move to mine and gather **frequent** itemsets from our training dataset (`order_products_train.csv`). In other words, we will focus on finding regularities in the shopping behavior of customers, based on what `product` orders (a.k.a. transactions) are composed of.

Vocabulary

Let us define some basic vocabulary we will be tremendously be using:

- Let I be a set of items. An itemset is a subset of I.
- Let D be a transaction database such that each transaction is an itemset.
- Frequency of an itemset is the number of transactions including the itemset.
- For a given support, an itemset is said to be frequent if its frequency is no less than the support.

SPMF Open-Source Library

Many (if not all) algorithms tested upon this point will be implementations from SPMF library (http://www.philippe-fournier-viger.com/spmf/ (http://www.philippe-fournier-viger.com/spmf/)); an open-source data mining mining library written in Java, specialized in pattern mining.

A NodeJS JS/TypeScript wrapper, `class SPMF` for SPMF (spawning Java commands) has been written in order to execute these, while still exploiting the results with Typescript. Sources may be find in file `./src/class/spmf.ts` . Examples of its use are shown below.

Apriori

Keeping our goal in mind, we'll start by using the **Apriori Algorithm**. The object of this algorithm is to identify association between different sets of data, and to find out patterns in data using a "bottom up approach", as frequent subsets are extended one item at a time; and is based on a property called the anti-monotone property: It states that if an item set is not frequent, then none of its supersets can be frequent. As a result, the list of potential frequent itemset gets smaller as mining progresses [1].

Though the Apriori Algorithm is easy to implement and understand, it is far from being performant the reasons stated above.

Dataset formatting

Upon this point we will be using SPMF library's Java implementation of Apriori (through command lines), in order to mine frequent item sets from our dataset. This implementation needs the input data to be formatted as such:

- An item is represented by a positive integer.
- A transaction is a line in the file.
- In each line (transaction), items are separated by a single space.
- It is assumed that all items within a same transaction (line) are sorted according to a total order and that no item can appear twice within the same line.

For example, an input file is defined as follows:

```
1 2 5
2 7
```

Transforming this dataset into this format is pretty straightforward using the code we already wrote previously (same code, but encapsulated in a `CSVParser` class). To save us some time, already formatted data will be available to parse (file `custom_data/formatted-itemsets.csv`). Here is an example of `CSVParser` though, running on a custom 6-line dataset:

In [7]:
```
// Our custom parser class, looks like Jupyter/iTypescript doesn't like ES6 imports of custom Typescript classes.
import * as CSVParser from './dist/class/csv-parser.class.js';
import { ProductOrder } from './src/interface/product-order.interface';
import { Group } from './src/interface/group.interface';

function formatData(): void {
    $$async$$ = true;
    console.log('Generating transactions, this might take a while...');

    let groups: Group<ProductOrder,number>[] = [];

    new CSVParser.CSVParser<ProductOrder>(`custom_data/small_order_products__train.csv`)
        // Grouping items by order_id, and mapping every item composing these itemsets to their product_id.
        .generateItemsets<number>('order_id', (productOrder: ProductOrder) => Number.parseInt(productOrder.product_id))
        // Once execution is complete, writing the formatted dataset into a proper file.
        .finally( () => {
            // Writing number of product per order_id in a new file : The array of already formatted rows is joined by a return carriage character.
            console.log('Example of output:');
            console.log(groups.join('\r\n'));
            $$done$$();
        })
        // On group reception, formatting the items composing is as a ROW (joined by plain space character), and pushing it the output array.
        .subscribe((group: Group<ProductOrder,number>) => groups.push(group.items.sort( (a: number, b: number) => a - b ).join(' ')) )
}

formatData();

```

```
Generating transactions, this might take a while...
Example of output:
1
101 1002
10 100 2000
```

Parsing the results

This Apriori implementation enlights us with multiple information other than the frequent itemsets themselves, as it outputs multiple results through both `stdout` , including:

- The number of candidates;
- The maximum size of candidates the algorithm stopped at;
- The frequent itemsets count;
- Maximum memory usage;
- Total execution time.

Mined frequent itemsets may be find in the specified output file, formatted as below:

```
...A #SUP: B

-- EXAMPLES OF OUTPUT --
49628 #SUP: 186
49683 #SUP: 2413
13176 21137  #SUP: 164
13176 21903  #SUP: 175
```

Where `...A` represents the spread of `item_ids` composing the frequent itemset; and `B` the support of this itemset (in term of number of occurency).

Eventhough the output format is not rigorously `.csv` friendly, we can still easily parse the output file with our TypeScript application with no addtionnal dependency or need of code; using a little hack, defining the string `#SUP:` as a `.csv` separator. This will result in us parsing an Array of 2-item rows:

```
[ ...item_ids, support ]
```

This is own our homemade SPMF wrapper inherently works.

EDIT: V2 of SPMF Wrapper uses Regular Expressions, see below

Last step will trivially be to parse those item_ids and joining these results with the `products.csv` table for enhanced visualisation.

Maximum support

In the extend of running the Apriori Algorithm, we need to define a proper minimum support value, as defined above. This is a crucial step, as setting the support to an exagerated value would result no itemsets to be find; while setting it to a too low value will inherently result in a very long execution time.

As seen previously, `Banana` is the most frequent item in out dataset, being recorded in 18,726 of the 131,209 transactions i.e. 14.3% of all the orders. This number also represents the maximum support of any itemset in our dataset, as being the support of itemset composed of the `Banana` singleton.

First runs

Let us run Apriori a few times, with a different minimum support threshold, in order to compare the execution time as well as the number of return candidates. One small drawback of this implementation being that it returns single-item candidates (which are not of interest at all), we'll also specify the number on multiple-items candidates found.

In [8]:
```
import * as SPMF from './dist/class/spmf.class.js';
import { SPMFResults, ItemSet, Rule } from './src/class/spmf.class';

function aprioriRun(): void {
    $$async$$ = true;
    console.log('Mining patterns, this might take a while...');
    // Our custom SPMF wrapper
    new SPMF.SPMF('Apriori')
        // Loading from file
        .fromFile(`custom_data/formatted_itemsets.txt`)
        // Executes Apriori with 5% support
        .exec<ItemSet>(5)
        // Listening for results
        .subscribe((results: SPMFResults<ItemSet>) => {
            // Wrapper returns both the stats...
            console.log('Stats:');
            console.log(results.stats);
            // ... and the frequent itemsets. Showing the first two itemsets:
            console.log('First two frequent itemsets:');
            console.log(results.output.slice(0,2));

            $$done$$();
        });
}

aprioriRun();
```

```
Mining patterns, this might take a while...
Stats:
{ candidates: 28,
  executionTime: 456,
  memory: 59.766456604003906 }
First two frequent itemsets:
[ { support: 15480, items: [ '13176' ] },
  { support: 10894, items: [ '21137' ] } ]
```

Minimum support	Number of candidates	Frequent itemsets count	Multiple-items itemsets	Execution time
13%	1 (Banana, as expected)	1	0	422 ms
7%	10	4	0	423 ms
3%	153	17	0	748 ms
1%	5,467	120	14	8.2 s
0.5%	33,042	364	108	48.9 s
0.3%	210,892	1125	478	4 mn 58 s

As expected, the speed of Apriori gets in the way pretty fast when it comes to parsing a dataset of such size, and with a maximum support so low. In order to find more relevant itemsets, we may have to transform the dataset in order reduce the overall total number of candidates.

Our first frequent itemsets

Results of Apriori runs may be find in folder `/output`, labelled as such: `output_approri_#SUPPORT.txt`. Lets have a look to multiple-itemed frequent itemsets Apriori has gathered, for a minimum support of 0.3%. We want to display our itemsets through a Chord Chart (https://www.zingchart.com/docs/chart-types/chord-diagrams/) using `zingchart`.

With this intent, we need to format the data, `zing-chart` -way, as such:

```
{
    name: 'Banana',
    values '0, 1'
}, {
    name: 'Avocado',
    values '1, 0'
}
```

... where 1 is the support of the itemset `<{'Banana','Avocado'}>`.

Such a transformation is quite tedious to explain, but I believe it's easy to understand line-by-line, though the code below (if not, I do apologize for any headache I may have cause).

Note that a class representing a Chord node has been created, in which you can easily manage relations of a serie of nodes. Code is available in the source codes `./src/class/chord-serie.class.ts`.

In [9]:

```
import * as SPMF from './dist/class/spmf.class.js';
import * as CSVParser from './dist/class/csv-parser.class.js';
import * as ChordSerie from './dist/class/chord-serie.class.js';
import * as join from './dist/function/join.function.js'

function parseAprioriResults(): void {
    $$async$$ = true;
    console.log('Parsing Apriori results, this might take a while...');

    // Loading the 'product.csv' table.
    new CSVParser.CSVParser<Product>(`instacart_basket_data/products.csv`).loadAll()
    .then( (products: Product[]) => {
        console.log('Finished loading products');

           // Our custom SPMF wrapper
        new SPMF.SPMF('Apriori')
        .loadResultsFromFile<ItemSet>('output/output_apriori_03.txt')
        .subscribe((results: SPMFResults<ItemSet>) => {
            results.output = results.output
                // Only 2-item itemsets
                .filter( (itemset: ItemSet) => itemset.items.length == 2 )
                // Sorted, highest support first
                .sort( (a: ItemSet, b: ItemSet) => b.support - a.support)
                // Only the 20 most frequent itemsets
                .slice(0,20);

            // Gathering all items composing the itemsets.
            let uniqueItems: string[] = [].concat.apply([], results.output.map( (itemset: ItemSet) => itemset.items ) )
                // Filtering unique items.
                .filter( (itemId: string, index: number, that: string[]) => that.indexOf(itemId) === index );

            // Mapping unique ids to a ChordSerie node.
            let series: ChordSerie.ChordSerie[] = uniqueItems.map( (uniqueItemId: string) => new ChordSerie.ChordSerie(uniqueItemId) );

            // Now we need to populate each node, lets start by noticing all nodes of the size of the array, so they can create inner arrays of such size.
            series.forEach( (serie: ChordSerie.ChordSerie) => serie.setSeriesNumber( series.length ) )

            // Then for each itemset, we need to populate each node
            results.output.forEach( (itemset: ItemSet) => {
                // So for each items composing the itemset...
                itemset.items.forEach( (itemId: string, index: number, that: string[]) => {
                    // Copying array just to be sure, shouldn't be a problem though.
                    let copy: string[] = that.slice(0);
                    // Deleting the item id we're currently parsing
                    copy.splice(index,1);
                    // Creating an array of all the nodes related to this item id
                    let indexesOfRelatedNodes: number[] = copy.map( (id: string) => uniqueItems.indexOf(id) )

                    // Noticing the node of its relations
                    series[uniqueItems.indexOf(itemId)].addRelation(itemset.support, indexesOfRelatedNodes);
                })
            });

            let chartOptions: string = JSON.stringify({
                id: 'aprioriItemSetChart',
                data: {
                    type:'chord',
                    title: {
                        text: 'Top 20 most frequent 2-items itemsets'
                    },
                    legend: {
                        overflow: 'scroll',
                        layout: 'vertical',
                        width: 200
                    },
                    options: {
                        'angle-padding': 1,
                        'band-width': 10,
                        'band-space': 5,
                        'radius': 190,
                        style: {
                            band: { },
                            //chord: { 'border-width': 0, flat: true },
                            tick: { visible: false },
                            item: { visible: false },
                            label: { visible: false }
                        },
                    },
                    tooltip:{},
                    // Populating with product names
                    series: series.map( (serie: ChordSerie.ChordSerie[]) => serie.getSerie( (id) => join.join<Product>(products, 'product_id', id, 'product
                }
            });

            $$.html(`
                <div id="aprioriItemSetChart"></div>
                <script>
                  zingchart.render(${chartOptions});
                </script>
            `);

            $$done$$();
        });
    });
}

parseAprioriResults();
```

```
Parsing Apriori results, this might take a while...
Finished loading products
```

Out[9]:

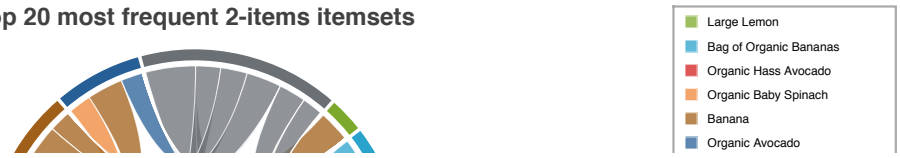


Chart above has been populated with the 20 most popular 2-item itemsets mined by Apriori with a minimum support of 0.3%; the chords of the chart reprensenting the itemsets, of which you can have the support by hovering the graph with your mouse.

It gives a visual idea of which items are often bought together (featuring the itemsets and their support through the chords), as well as the most popular items within these itemsets.

It is worth noticing that only 13 products share all top 20 most frequent 2-itemed itemset sets.

Fruits and vegetables appears to be over represented within the frequent itemsets, which makes sense, as fruits and vegetables are very often the products we buy altogether when going to the supermarket. Yet within the aisle, from a marketing point of view, it could be interesting to organise the products such as the most popular items are apart each other. One may be tempted to buy `Apples` on his way to find `Strawberries` after weightening his freshly bought `Bananas` ; most frequent 2-item itemset being `<{"Banana","Organic Strawberries"}>` .

In addition, 3 itemsets have also be mined, and it can be of interest to take a glance at them:

In [10]:
```
import * as SPMF from './dist/class/spmf.class.js';
import * as CSVParser from './dist/class/csv-parser.class.js';
import * as join from './dist/function/join.function.js'

function threeItemSets(): void {
    $$async$$ = true;
    console.log('Parsing Apriori results, this might take a while...');

    // Loading the 'product.csv' table.
    new CSVParser.CSVParser<Product>(`instacart_basket_data/products.csv`).loadAll()
    .then( (products: Product[]) => {
        console.log('Finished loading products');

           // Our custom SPMF wrapper
        new SPMF.SPMF('Apriori')
        .loadResultsFromFile<ItemSet>('output/output_apriori_03.txt')
        .subscribe((results: SPMFResults<ItemSet>) => {
            let rows: string = results.output
                // Only 2-item itemsets
                .filter( (itemset: ItemSet) => itemset.items.length > 2 )
                // Sorted, highest support first
                .sort( (a: ItemSet, b: ItemSet) => b.support - a.support)
                // Creating HTML Rows
                .map( (itemset: ItemSet) => {
                    let items: string[] = itemset.items.map( (id: string[])  => {
                        return `<td>${join.join<Product>(products, 'product_id', id, 'product_name')}</td>`
                    });

                    return `<tr>${items.join('')}<td>${itemset.support}</td></tr>`;
                })
                // Only the 10 most frequent itemsets
                .slice(0,10);

            $$.html(`
                <h3>Top 10 3-items frequent itemsets</h3>
                <table style="width:100%">
                  <tr>
                    <th>Item 1</th>
                    <th>Item 2</th>
                    <th>Item 3</th>
                    <th>Support</th>
                  </tr>
                  ${rows.join('')}
                </table>
            `);

            $$done$$();
        });
    });
}

threeItemSets();
```

```
Parsing Apriori results, this might take a while...
Finished loading products
```

Out[10]:

Top 10 3-items frequent itemsets

Item 1	Item 2	Item 3	Support
Bag of Organic Bananas	Organic Strawberries	Organic Hass Avocado	710
Bag of Organic Bananas	Organic Strawberries	Organic Raspberries	649
Bag of Organic Bananas	Organic Strawberries	Organic Baby Spinach	587
Bag of Organic Bananas	Organic Raspberries	Organic Hass Avocado	531
Bag of Organic Bananas	Organic Baby Spinach	Organic Hass Avocado	497
Organic Baby Spinach	Banana	Organic Avocado	484
Banana	Large Lemon	Organic Avocado	477
Banana	Limes	Large Lemon	452
Bag of Organic Bananas	Organic Strawberries	Organic Cucumber	424
Limes	Large Lemon	Organic Avocado	389

It's pretty fun to see that the top 5 (6 out of top 10) of most commonly bought-together products are all itemsets of **organic** products. This really represents the new trend and consumer habit of buying only organic products: One having in mind to buy organic products will only buy organic products. This behaviour result in creating interesting frequent datasets!

These frequent itemsets clealy points towards associations rules stating `If one buys 2 organic products, he will buy 1 other organic products` with a very high confidence. Let's keep this idea in mind for later.

LCM Algorithm

LCM is an algorithm know to be the fastest for mining frequent **closed** itemsets; knowing that an itemset X is closed in a dataset S if no proper super-itemset Y that has the same support count as X in S exists. Main reason of mining closed frequent datasets over all datasets is that such a set is usually much smaller than the set of frequent itemsets, eventhough no information is lost, as the entire set can be regenerated from the closed set[2].

We will be using SPMF library's JAVA implementation of LCM, in order to mine frequent closed itemsets from our dataset; An advantage beingi that this implementation inputs the same file format as the Apriori Algorithm of the same library, thus, there is no need to format our dataset in a different manner.

Running the algorithm

It would be interesting to compare the performances and results of the LCM and Apriori algorithms; thus, let's run LCM over the same dataset again, and with the same support. You can also try it by yourself by executing the code below:

In [11]:
```
import * as SPMF from './dist/class/spmf.class.js';

function lcmRun(): void {
    $$async$$ = true;
    console.log('Mining patterns, this might take a while...');
    // Our custom SPMF wrapper
    new SPMF.SPMF('LCM')
        // Loading from file
        .fromFile(`custom_data/formatted_itemsets.txt`)
        // Executes LCM with 5% support
        .exec<ItemSet>(5)
        // Listening for results
        .subscribe((results: SPMFResults<ItemSet>) => {
            // Wrapper returns both the stats...
            console.log('Stats:');
            console.log(results.stats);
            // ... and the frequent itemsets. Showing the first two itemsets:
            console.log('First two frequent itemsets:');
            console.log(results.output.slice(0,2));

            $$done$$();
        });
}

lcmRun();
```

```
Mining patterns, this might take a while...
Stats:
{ candidates: undefined, executionTime: 409, memory: undefined }
First two frequent itemsets:
[ { support: 15480, items: [ '13176' ] },
  { support: 10894, items: [ '21137' ] } ]
```

Minimum support	Frequent closed itemsets count	Multiple-items itemsets	Execution time
13%	1	0	204 ms
7%	4	0	275 ms
3%	17	0	328 ms
1%	120	16	1.2 s
0.5%	364	108	2.3 s
0.3%	1125	339	4.9 s
0.1%	4444		24.4 s

Let's reduce the minimum support even more !

Enhanced perfomances of LCM over Apriori allows us to reduce the minimum even more while mining patterns on our dataset with decent executing times (See table above, with an execution for a minimum support of .1%). This allowed us to find some quite revealant 4-items itemsets, with still a decent support value:

In [12]:

```
import * as SPMF from './dist/class/spmf.class.js';
import * as CSVParser from './dist/class/csv-parser.class.js';
import * as join from './dist/function/join.function.js'

function lcmItemsets(): void {
    $$async$$ = true;
    console.log('Mining patterns, this might take a while...');

    new CSVParser.CSVParser<Product>(`instacart_basket_data/products.csv`).loadAll()
    .then( (products: Product[]) => {
        console.log('Finished loading products');

        // Our custom SPMF wrapper
        new SPMF.SPMF('LCM')
            // Loading from file
            //.fromFile(`custom_data/formatted_itemsets.txt`)
            // Executes LCM with .1% support
            //.exec<ItemSet>(.1)
            .loadResultsFromFile<ItemSet>('output/output_lcm_01.txt')
            // Listening for results
            .subscribe((results: SPMFResults<ItemSet>) => {
                let rows: string = results.output
                    // Only 2-item itemsets
                    .filter( (itemset: ItemSet) => itemset.items.length > 3 )
                    // Sorted, highest support first
                    .sort( (a: ItemSet, b: ItemSet) => b.support - a.support)
                    // Creating HTML Rows
                    .map( (itemset: ItemSet) => {
                        let items: string[] = itemset.items.map( (id: string[])  => {
                            return `<td>${join.join<Product>(products, 'product_id', id, 'product_name')}</td>`
                        });

                        return `<tr>${items.join('')}<td>${itemset.support}</td></tr>`;
                    })
                    // Only the 10 most frequent itemsets
                    .slice(0,10);

                $$.html(`
                    <h3>4-items frequent itemsets</h3>
                    <table style="width:100%">
                      <tr>
                        <th>Item 1</th>
                        <th>Item 2</th>
                        <th>Item 3</th>
                        <th>Item 4</th>
                        <th>Support</th>
                      </tr>
                      ${rows.join('')}
                    </table>
                `);

                $$done$$();
            });
    });
}

lcmItemsets();
```

```
Mining patterns, this might take a while...
Finished loading products
```

Out[12]:

4-items frequent itemsets

Item 1	Item 2	Item 3	Item 4	Support
Bag of Organic Bananas	Organic Strawberries	Organic Raspberries	Organic Hass Avocado	228
Bag of Organic Bananas	Organic Strawberries	Organic Baby Spinach	Organic Hass Avocado	163
Banana	Limes	Large Lemon	Organic Avocado	146
Bag of Organic Bananas	Organic Strawberries	Organic Cucumber	Organic Hass Avocado	140
Organic Baby Spinach	Banana	Large Lemon	Organic Avocado	134

These results points in the same way than our previous observations: organic products tends to be bought alltogether.

Quick comparison of LCM and Apriori

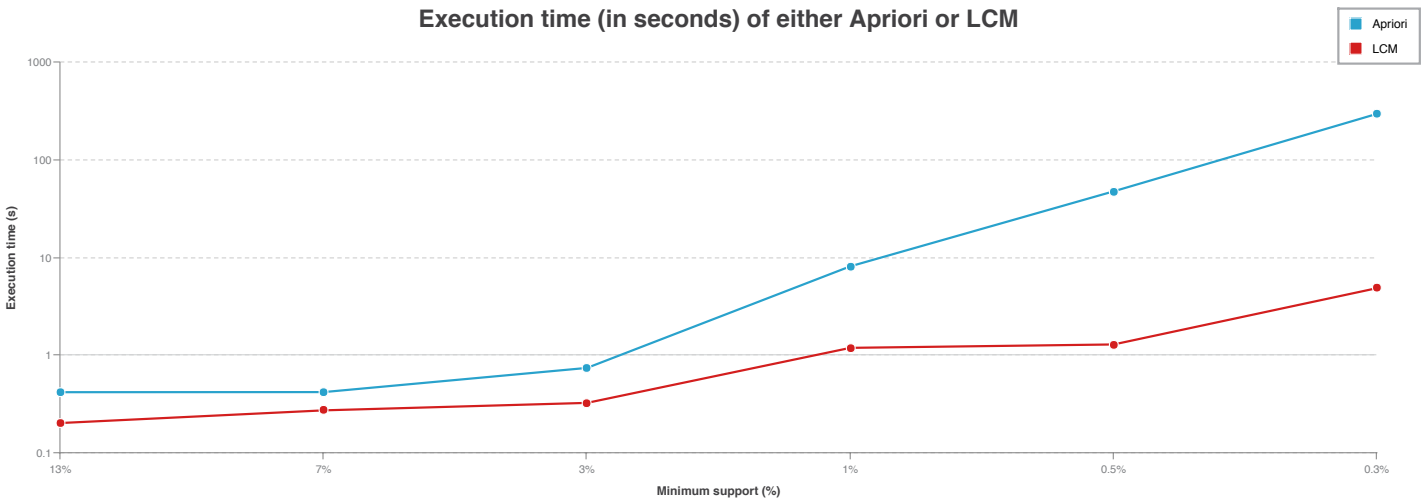
Let's have a quick look over the behaviours and execution of both tested algorithms:

Minimum support	Apriori execution time	LCM execution time
13%	422 ms	204 ms
7%	423 ms	275 ms
3%	748 ms	328 ms
1%	8.2 s	1.2 s
0.5%	48.9 s	1.3 s
0.3%	4 mn 58s	4.9 s

In [13]:

```
function executionTimeChart(): void {

    let chartOptions: string = JSON.stringify({
        id: 'executionTimeChart',
        data: {
            type: 'line',
            scaleY: {
                progression: 'log',
                logBase: 10,
                label: {
                  text: "Execution time (s)"
                },
                item: {
                  fontSize: 10
                },
            },
            scaleX: {
                values: ['13%','7%','3%','1%','0.5%','0.3%'],
                label: {
                  text: "Minimum support (%)"
                },
                item: {
                  fontSize: 10
                },
            },
            title: {
                text: 'Execution time (in seconds) of either Apriori or LCM'
            },
            legend: {

            },
            series: [
                { values: [.422,.423,.748,8.2,48,298], text: 'Apriori' },
                { values: [.204,.275,.328,1.2,1.3,4.9], text: 'LCM' }
            ]
        }
    });

    $$.html(`
        <div id="executionTimeChart"></div>
        <script>
          zingchart.render(${chartOptions});
        </script>
    `);

};

executionTimeChart();
```

Out[13]:



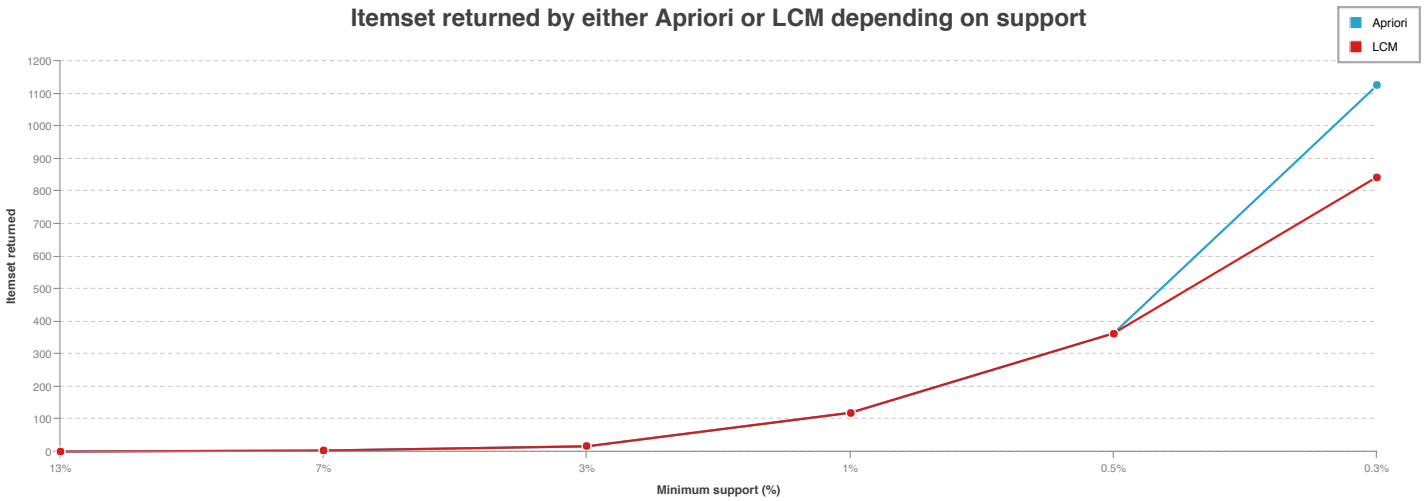
In terms of performance, LCM crushed Apriori when it comes to lower minimum support. This is anything but a surprise, as Apriori is nowadays outdated, as faster and more memory efficient algorithms for mining `all datasets` have been proposed since, like `FPGrowth` which we will use later. In the meantime, LCM is known to be one of the most efficient algorithm for mining closed frequent itemsets.

Minimum support	Apriori itemsets count	LCM itemsets count
13%	1	1
7%	4	4
3%	17	17
1%	120	120
0.5%	364	364
0.3%	1125	844

In [14]:

```
function itemSetChart(): void {

    let chartOptions: string = JSON.stringify({
        id: 'itemSetChart',
        data: {
            type: 'line',
            scaleY: {
                label: {
                  text: "Itemset returned"
                },
                item: {
                  fontSize: 10
                },
            },
            scaleX: {
                values: ['13%','7%','3%','1%','0.5%','0.3%'],
                label: {
                  text: "Minimum support (%)"
                },
                item: {
                  fontSize: 10
                }
            },
            title: {
                text: 'Itemset returned by either Apriori or LCM depending on support'
            },
            legend: {

            },
            series: [
                { values: [1,4,17,120,364,1125], text: 'Apriori' },
                { values: [1,4,17,120,364,844], text: 'LCM' }
            ]
        }
    });

    $$.html(`
        <div id="itemSetChart"></div>
        <script>
          zingchart.render(${chartOptions});
        </script>
    `);

};

itemSetChart();
```

Out[14]:



In terms of returned itemsets, LCM globally returns let itemsets than Apriori. This was to be expected given that those Algorithms don't output the same set of itemsets; Apriori returning all itemsets while LCM only returns **closed itemsets**. However it may be of interest to notice that both algorithms return the **same** result upon a certain minimum support given the fact that itemsets returned are closed no matter what algorithm is used.

Frequent item sets association rules

Once patterns have been mined, we can also determine association rules, as each item may have several relations with others. These relations thus indirect relationships between the items composing the transactions; expressed with a certain **confidence**, as an example:

```
Diapers implies beer with 65% confidence
```

FPGrowth association mining

Association rules mining generally consist in two steps: the first step being to discover frequent itemsets (With Apriori, LCM or any other algorithm as we did previously); and the second, to generate rules by using these frequent itemsets.

We'll be using in that order yet another all-in-one Algorithm from the SPMF Library, the `FPGrowth_association_rules` algorithm, which basically mine item sets using a faster, more memory efficient algorithm than Apriori: `FPGrowth` ; before generating the association rules.

Algorithm takes 2 input values:

- The minimum support of mined itemsets, as before;
- The minimum confidence of resulting association rules;

Parsing the results

Parsing the results may defer a bit, as the output file now indicates rules instead of plain itemsets, the support of the association, as well as the confidence of the rule.

Example of output:

```
1 ==> 2 4 5 #SUP: 3 #CONF: 0.75
5 6 ==> 1 2 4 #SUP: 3 #CONF: 0.6
4 7 ==> 1 #SUP: 3 #CONF: 0.75
```

Such a file would result in being parsed the following way with our current code:

```
['1 ==> 2 4 5', '3 #CONF: 0.75']
```

Nothing to worry about though, as a really hacky to do it would be to `split()` both resulting strings using separators `==>` and `#CONF`. I'd really wish I would have use Regular expressions at this point though...

Late update: Regular expressions

That was I ended up doing eventually, keeping in mind that we will need to parse outpit files from even more algorithms (sequential pattern mining, etc...). I'm thus using the following Regular expressions to match results as they are gatheresd from output files; and using even more regular expressions within the matches of expressions to parse for numeric values.

```
Matching itemsets:
/^(\d* )+#SUP: (\d*( )*)+\t*\n*\r*$/g

Matching association rules:
/^(\d* )+==> (\d* )+#SUP: (\d* )+#CONF: (\d*)+\.*\d*\t*\n*\r*$/g
```

Running the algorithm

You can try the algorithm by yourself, by editing and executing the code below:

In [15]:
```
import * as SPMF from './dist/class/spmf.class.js';

function associationRulesMining(): void {
    $$async$$ = true;
    console.log('Mining association rules, this might take a while...');
    // Our custom SPMF wrapper
    new SPMF.SPMF('FPGrowth_association_rules')
        // Loading from file
        .fromFile(`custom_data/formatted_itemsets.txt`)
        .exec<Rule>(0.1,25)
        // Listening for results
        .subscribe((results: SPMFResults<Rule>) => {
            // Wrapper returns both the stats...
            console.log('Stats:');
            console.log(results.stats);
            // ... and the frequent rules. Showing the first two rules:
            console.log('First two rules mined:');
            console.log(results.output.slice(0,2));

            $$done$$();
        });
}

associationRulesMining();
```

```
Mining association rules, this might take a while...
Stats:
{ candidates: undefined, executionTime: 54, memory: undefined }
First two rules mined:
[ { support: 198,
    confidence: 0.3168,
    items: [ '45' ],
    results: [ '24852' ] },
  { support: 136,
    confidence: 0.3919308357348703,
    items: [ '46149' ],
    results: [ '196' ] } ]
```

Association rules & Aisles relations

Itemset association rules are great candidates to Chord Diagram representation. We'll use these mined associations to determine how aisles interact with each other. The idea we have in mind is simple: We will map every item and result (which are `product_ids`) of each rule to their `aisle_id`, in order to express associations rules as such:

`If customer buys a product from aisle A (and B) then he will buy a product from aisle C (and D)`

From a marketer point of view, we want aisle A (and B) as far as possible from aisle C (and D)... Thus when parsing our newly parsed rules, keeping our example in mind: Each time we encounter id of `Aisle A`, we will increment the value linked to the band (a.k.a. edge) between it and `Aisle C`.

The resulting graph will show the weigted relations between each aisle.

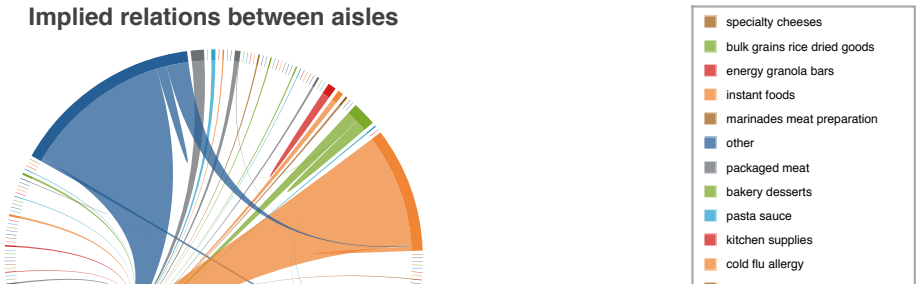
Note that we will not take `confidence` into account. Given the fact that minimal confidence as been set to 25%, we believe confidence value should not be a discriminative factor for mapping aisles (probability of each is high).

In [16]:

```
import * as SPMF from './dist/class/spmf.class.js';
import * as CSVParser from './dist/class/csv-parser.class.js';
import * as ChordSerie from './dist/class/chord-serie.class.js';
import * as join from './dist/function/join.function.js'

function parseAssociationRules( ignoreIds: string[] = [] ): void {
    $$async$$ = true;
    console.log('Parsing association rules, this might take a while...');

    let products: Product[] = [];
    let aisles: Aisle[] = []

    // Loading the 'product.csv' table.
    new CSVParser.CSVParser<Product>(`instacart_basket_data/products.csv`).loadAll()
    .then( (p: Product[]) => {
        products = p;
        console.log('Finished loading products');

        // Loading the 'aisles.csv' table.
        return new CSVParser.CSVParser<Aisle>(`instacart_basket_data/aisles.csv`).loadAll()
    })
    .then( (a: Aisle[]) => {
        aisles = a;
        console.log('Finished loading aisles');

           // Our custom SPMF wrapper
        new SPMF.SPMF('FPGrowth_association_rules')
        .loadResultsFromFile<Rule>('output/output_fpgrowth_assoc_01_25.txt')
        .subscribe((results: SPMFResults<Rule>) => {
            // Gather all the aisle_ids
            let aislesIds: number[] = aisles.map( (aisle: Aisle) => aisle.aisle_id );

            // Series will be aisles, as planned.
            let series: ChordSerie.ChordSerie[] = aisles.map( (aisle: Aisle) => new ChordSerie.ChordSerie(aisle.aisle_id).setSeriesNumber(aisles.length) )

            // For each rule...
            results.output.forEach( (rule: Rule) => {
                // Map all aisle_ids the rule is pointing towards.
                let indexesOfResults: number[] = rule.results.map( (id: string) => aislesIds.indexOf( join.join<Product>(products, 'product_id', id, 'aisl

                // For each item composing the rule
                rule.items.forEach( (id: string) => {
                    // Get the aisle_id from the product_id
                    let aisleId: number = join.join<Product>(products, 'product_id', id, 'aisle_id');
                    // If it's ignored, then meh.
                    if(ignoreIds.includes(aisleId)) return;

                    // Else notice the corresponding aisle it has a relation with others.
                    let index: string = aislesIds.indexOf( aisleId );
                    series[index].addRelation(1,indexesOfResults);
                });
            })

            let chartOptions: string = JSON.stringify({
                id: 'associationRulesChart' + ignoreIds.join('_'),
                data: {
                    type:'chord',
                    title: {
                        text: 'Implied relations between aisles',
                    },
                    subtitle: {
                        text: ignoreIds.length ? `Aisles ignored : ${ignoreIds.join(' ')}` : ''
                    },
                    legend: {
                        overflow: 'scroll',
                        layout: 'vertical',
                        width: 200
                    },
                    options: {
                        'angle-padding': 1,
                        'band-width': 10,
                        'band-space': 0,
                        'radius': 190,
                    "style":{
                        "chord":{
                            "border-width":0,
                            "flat":false
                        },
                        "band":{ "border-width": 0 },
                        "tick":{ "visible": false },
                        "item":{ "visible":false },
                        "label":{ "visible": false }
                    }
                    },
                    tooltip:{},
                    // Populating with product names
                    series: series.map( (serie: ChordSerie.ChordSerie[]) => serie.getSerie( (id) => join.join<Aisle>(aisles, 'aisle_id', id, 'aisle') ) ),
                }
            });

            $$.html(`
                <div id="associationRulesChart${ignoreIds.join('_')}"></div>
                <script>
                  zingchart.render(${chartOptions});
                </script>
            `);

            $$done$$();
        });
    });
}

parseAssociationRules();
```

```
Parsing association rules, this might take a while...
Finished loading products
Finished loading aisles
```

Out[16]:



Graph above enlights us with many interesting information:

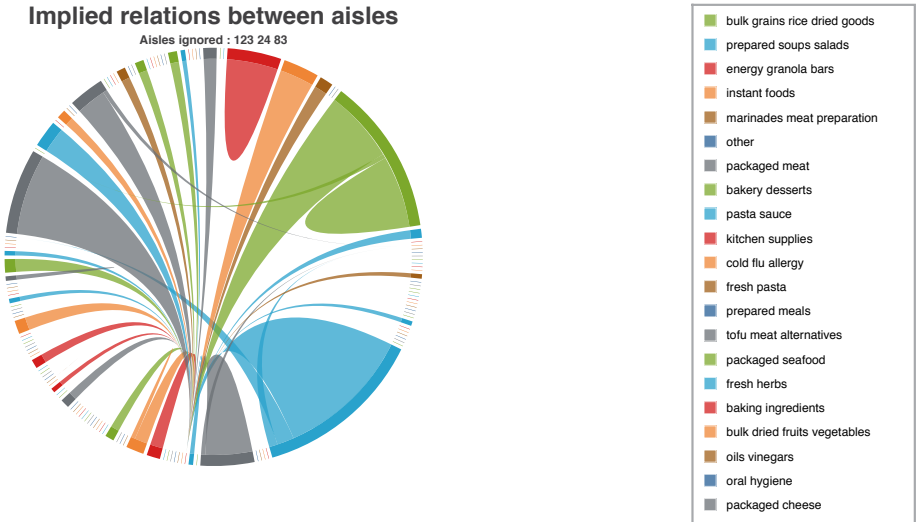
- You want to keep `fresh vegetables`, `packed vegetables fruits` and `fresh fruits` aisles as far as possible from each other (which is definitely a difficult task I concede);
- Many rules associate `fresh fruits` products to products of itself (as we saw previously): You will have to deal inner aisle organisation !

Let's ignore aisles `fresh vegetables`, `packed vegetables fruits` and `fresh fruits` relations and recompile the graph:

In [21]:
```
1 parseAssociationRules(['123','24','83']);
```

```
Parsing association rules, this might take a while...
Finished loading products
Finished loading aisles
```

Out[21]:



Many outer relations of associations from other aisles still point toward `fresh fruit` aisle products. *omnibus viis Romam pervenitur*.

Though we can still glance some interesting things happening in `yoghurt` and `sparkling water aisles`, as it seems products from these aisles are often bought together ! If it actually definitely sounds silly that people want to buy different kind of sparkling water at the same time.

As a general conclusion to itemset pattern mining... Never have short storage of bananas, and just put the fresh fruits aisle all across the whole store :)

Sequential pattern mining

Sequential pattern mining consists of discovering interesting subsequences in a set of sequences (where, has a comparison, mining frequent itemsets consisted in discovering itemsets in a set of transaction). In the context of our example, sequential pattern mining can be used to find the sequences of items frequently bought by customers, as we acknowledged that transaction data could have be ordered by time, with the proper table joins (`orders.csv`). This can be useful to understand the behavior of customers to take marketing decisions.

Such a transformation is however pretty heavy, and an already transformed dataset, `output/transactions_seq.txt` is already provided with this notebook; a 4-columns csv-like file (separators are tabulation characters) as such:

- Column 1: user id;
- Column 2: order number (not an order id, but a number in the sequence of orders);
- Column 3: size of order;
- Column 4: list of products in text format (without spaces), separated by a comma;

In order to mine sequential pattern from our data, we will use the CloSpan Algorithm, from the SPMF library. This algorithm is used for discovering **closed** sequential patterns in sequence databases. As a closed frequent itemset is to a frequent itemset; a closed sequential pattern is a sequential pattern such that it is not strictly included in another pattern having the same support.

Data formatting

The input file format for SPMF CloSpan is defined as follows:

- Each line represents a sequence from a sequence database, and ends with the value -2;
- Each item from a sequence is a positive integer, and items from the same itemset within a sequence are order ascendingly, and separated by single space;
- Each itemset is separated by the value -1;

Example of input from the SPMF Documentation :

```
1 -1 1 2 3 -1 1 3 -1 4 -1 3 6 -1 -2
1 4 -1 3 -1 2 3 -1 1 5 -1 -2
5 6 -1 1 2 -1 4 6 -1 3 -1 2 -1 -2
5 -1 7 -1 1 6 -1 3 -1 2 -1 3 -1 -2
```

Back to our example, the file `transactions_sec.txt` has yet be found quite unusable, as the 'comma' has been used as a separator to separate items within the itemsets, making it impossible to map the `product_names` to their actual id.

```
96980    15    10
Asparagus,GreekExtraVirginOliveOil,Jelly,Blackberry,OrganicButterhead(Boston,Butter,Bibb)Lettuce,OrganicIcebergLettuce,RiceWhip,RipeLargePittedOlives
```

In the exemple above, `Jelly,Blackberry` and `OrganicButterhead(Boston,Butter,Bibb)Lettuce` are two separate products. Thus total count of products is 8, not 10.

Only way to compute this file has been to map all unique `strings` to a new integer (in a certain way, to compute our own `products.csv` (re-joining both `orders` and `order_product` is not an easy process). It's a bit sad though as we won't be able to compute more stats on aisles or departments from our results.

Here will be the transformations to apply :

- Group each order per `client_id` : This will be our sequences;
- Map the product to an integer;
- Order and format the sequence as specified.

Algorithm below executes this exact process, for you to tinker with. Two files have been generated and available for you though.

- `products_transactions_seq.csv` , a map of products to their 'new' unique id;
- `formatted_transactions_seq.txt` , the formatted `transactions_sec.txt` as intended.

Few interfaces had to be declaed too:

```
export interface TransactionSeq {
    user_id: string,
    order_number: string,
    order_size: string,
    products: string
}

export interface ItemSetSeq {
    productIds: number[],
    order: number
}
```

In [22]:

```
import * as CSVParser from './dist/class/csv-parser.class.js';
import { TransactionSeq } from './src/interface/transaction-seq.interface';
import { ItemSetSeq } from './src/interface/item-set-seq.interface';

function formatTransactionsSeq(): void {
    console.log('Gathering data, this might take a while...');
    $$aync$$ = true;

    let uniqueProducts: string[] = [];
    let sequences: string[] = [];

    // Open the transaction_sec file.
    new CSVParser.CSVParser<TransactionSeq>('custom_data/small_transactions_seq.txt', { delimiter: '\t', columns: true })
        // Group by user_id and map TransactionSeq to ItemSetSeq.
        .generateItemsets<ItemSetSeq>('user_id', (transactionSeq: TransactionSeq) => {
            // Parsing products for product_ids.
            let productIds: number[] = transactionSeq.products.split(',').map( (productName: string) => {
                // Looking for the productName.
                let index: number = uniqueProducts.indexOf(productName.replace('\r',''));
                // If the as already been incountered, returning its index.
                if(index > -1) return index;

                // Else adding it to our 'uniqueProducts' array.
                uniqueProducts.push(productName)
                return uniqueProducts.length - 1;
            });

            return {
                // Keep in mind that items must be sorted asc. in the itemsets.
                productIds: productIds.sort( (a: number, b: number) => a - b ),
                order: Number.parseInt(transactionSeq.order_number)
            };
        })
        // Once all the group are gathers and sequences formatted.
        .finally( () => {
            // Example of output
            console.log('Example of output:');
            console.log(sequences.join('\r\n'));


            setTimeout( () => $$done$$(), 300);
        })
        // Once one group is available.
        .subscribe( (group: Group<TransactionSeq,ItemSetSeq>) => {
            // Let's format groups to a sequence as specified for input
            let sequence: string = group.items
                // Within the groups, we need to sort ItemSetSeq by order asc.
                .sort( (a: ItemSetSeq, b: ItemSetSeq) => a.order - b.order )
                // Keep in mind that items are separated by a plain space character within the itemsets.
                .map( (itemSetSeq: ItemSetSeq) => itemSetSeq.productIds.join(' ') )
                // Each itemset is separated by a ' -1 ' value
                .join(' -1 ');

            // A sequence ends with ' -2'
            sequences.push(sequence + ' -2')
        });
}

formatTransactionsSeq();
```

```
Gathering data, this might take a while...
```

Out[22]:

```
undefined
```

```
Example of output:
0 1 2 3 4 -1 0 2 3 5 6 7 -1 2 3 7 8 9 -1 2 3 4 7 9 -1 2 3 5 7 9 10 11 12 -1 2 3 7 9 -1 2 3 6 7 9 -1 2 3 7 9 13 14 -1 2 3 7 9 13 14 -1 2 3 6 7 9 14 15 16 17 -2
```

Parsing output

The output file introduces yet another format as such:

- Each line being a frequent sequential pattern;
- Each itemset being separated by the value -1;
- Support of the sequence is indicated.

Nothing major though, it's only a matter of yet another Regular Expression to match the results with our current existing code:

Where looking for at least 1 ensemble of a spread of any length of positive integers `\d+` followed by '-1 ' `((\d+ )+-1 )+` , and then followed by '#SUP :' and a positive integer `#SUP: \d+` .

```
Regular expression:
((\d+ )+-1 )+#SUP: \d+
```

Running the algorithm

In [23]:

```
import * as SPMF from './dist/class/spmf.class.js';

function sequencesMining(): void {
    $$async$$ = true;
    console.log('Mining sequences, this might take a while...');
    // Our custom SPMF wrapper

    new SPMF.SPMF('CloSpan')
    .fromFile('custom_data/formatted_transactions_seq.txt')
    .exec<Sequence>(20)
    .subscribe((results: SPMFResults<Sequence>) => {
        // Wrapper returns both the stats...
        console.log('Stats:');
        console.log(results.stats);
        // ... and the frequent sequences. Showing the first two sequecnes:
        console.log('First two sequences mined:');
        results.output.slice(0,2)
        console.log(results.output.slice(0,2));

        $$done$$();
    });
}

sequencesMining();
```

```
Mining sequences, this might take a while...
Stats:
{ candidates: undefined, executionTime: 588, memory: undefined }
First two sequences mined:
[ { itemsets: [ [Array] ], support: 6084 },
  { itemsets: [ [Array], [Array] ], support: 4045 } ]
```

Sequential patterns

File `output/output_clospan_2.txt` provided contains mined sequential patterns with `CloSpan` and a minimum support of 2%. Let's see what we can find about them...

No time to finish visualisation :(

References

[1]: "Kumar, Dr K Ramesh & D, Usha. (2013). Frequent Pattern Mining Algorithm for Crime Dataset: An Analysis. INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH.". Avalaible on https://www.researchgate.net/publication/266498986_Frequent_Pattern_Mining_Algorithm_for_Crime_Dataset_An_Analysis (https://www.researchgate.net/publication/266498986_Frequent_Pattern_Mining_Algorithm_for_Crime_Dataset_An_Analysis)

[2]: "Philippe Fournier Viger. Mining Frequent Closed Itemsets using the LCM Algorithm. SPMF documentation.". Avalaible on http://www.philippe-fournier-viger.com/spmf/LCM.php (http://www.philippe-fournier-viger.com/spmf/LCM.php)