Major Project

COMP4121: Advanced Algorithms

Oscar L. Downing

z5114817

December 7, 2020

Contents

1 Introduction

This project involves using the Spectral Clustering algorithm to classify securities (aka shares) on the Australian Securities Exchange (ASX). The fundamentals of the companies attached to a ASX code are transformed into statistics which is used to create a fully connected graph structure used in the algorithm. The values are scraped from Yahoo finance. This is used to create a similarity score for each share. The shares inputted are partitioned into subsets, these subsets are formed based on the similarity rating of the shares. The algorithm stores the scores in a weighted adjacency matrix.

Using linear algebra the original matrix is transformed into one that is sparse, easier to compute and approximates the solution to the NP hard problem of clustering points. The transformed graph is then fed into a K-Means algorithm (using Lloyd's method) which clusters the matrix. The algorithm is written in Haskell which is a fast language that has expressiveness allowing for nice equation-like functions.

The core of the project is the spectral clustering algorithm, this is detailed in section 2, with implementation details in section 3. The project also involves a auxiliary script for gathering data, this is detailed in 4. Installing and running the project can be found in section 5. Reflections on the project can be found in 6.

2 Description of the Spectral Clustering algorithm

On the top level, the algorithm takes for input a list of pairs, a label and a score. The details of this construction is detailed in sections 4. From these data points data points, $A = \{a_1, ..., a_n\}$, an undirected weighted similarity graph is constructed, represented by an adjacency matrix, this process is detailed in section 2.1. Essentially, the adjacency matrix is a list of lists, the element being an Adjacency record type: Adj, which stores the weight and edge information.

k is an input as well on the top level, this is an integer for the desired number of clusters for clustering to produce. sig is another input which determines the "size" of the cluster. These two parameters are difficult to calculate intelligently, and require fine tuning with trail and error.

The method consists of three stages: the preprocessing stage 2.2, the decomposition stage 2.3, and then K Means stage 2.4. The preprocessing stage creates a sparse matrix, which allows for cheaper computation of the Eigen decomposition. The decomposition stage involves calculating the Eigenvalues and Eigenvectors which reveal the connectivity of the graph. And the K Means stage the is the final step, clustering the data points with the insight the eigenvectors provide.

The following pseudocode is a summary of the core algorithm (un-normalised spectral clustering):

Input: A set of data points $A = \{a_1, ..., a_n\}$, number k of clusters to construct.

- Construct a similarity graph as described in 2.1. Let W be its weighted adjacency matrix.
- Compute the un-normalised Laplacian L.
- Compute the k eigenvectors $e_1, ..., e_k$ of L which correspond to k smallest eigenvalues
- Let E be the matrix of size $n \times k$ containing the eigenvectors $e_1, ..., e_k$ as columns.
- For $i = 1, ..., n$ let y_i be the vector corresponding to the i'th row of E
- Cluster points $\{y_1, ..., y_n\}$ using the k-means algorithm into clusters $C_1, ..., C_k$

Output: Clusters $A_1, ..., A_k$ with A_i defined as $A_i = \{v_j : y_j \in C_i\}$

2.1 Similarity Graph

The goal of the similarity graph is to have a representation of the data that allows for grouping together vertices based on some "measure" function. This involves connecting all the pairs of vertices v_i and v_j for which the corresponding data points a_i and a_i have a strictly positive similarity.

This boils down to iterating through a list of lists and creating a `Adj` type (adjacency representation) for every combination of pairs of data points: they are first transformed into a Edge representation containing each data point represented as a Vertex, and their "score" is used to calculate a weight of that Edge, both the weight and the Edge are contained in a `Adj` type. The function that maps the "score" of the data point to a similarity rating, is designed to reflect local neighbourhood relationships:

$w_{ij} \ = \ \exp(-\frac{\|a_i - a_j\|^2}{2\sigma^2})$

Where a_i and a_j are the "scores" of the data points. Sigma represents the "size" of the neighbourhood, that is, how fast the similarity decreases as distance Increases. After this iteration, the graph is fully connected, this is the similarity graph of the data points and is represented with an adjacency matrix.

2.2 Preprocessing Stage

Now, a $n \times n$ similarity graph is constructed, where n is the number of data points, let $W \ = \ \{w_{ij} \ : \ 1 \leq \ i, j \leq \ n\}$ refer to the its weighted adjacency matrix. This stage involves performing linear transformations on the matrix W, transforming the weights and creating a sparse matrix, which results in cheaper (in terms of time) computation of the Eigen decomposition, detailed in section 2.3.

First, the Degree matrix D is calculated. This is a $n \times n$ diagonal matrix, such that for each vertex, along the diagonal the sum of all the adjacent vertices are stored.

$D \ = \ I \ \ d_i$

Where $d_i \ = \ \sum_{j=1}^{n} w_{ij}$

And I is the identity matrix, and W is the weighted adjacency matrix.

Secondly, the unnormalized Laplacian matrix is computed. This matrix is sparse.

$L \ = \ D \ - \ W$

2.3 Decomposition Stage

The next stage consists of computing k eigenvectors $e_1, ..., e_k$ corresponding to k smallest (and non negative) eigenvalues of the Laplacian matrix L.

The resulting matrix E, of size $n \times k$, is constructed with the eigenvectors $e_1, ..., e_k$ as columns.

For $i = \{1, ..., n\}$ let y_i be the vector corresponding to the i^{th} row of E.

The vectors $\{y_1, ..., y_n\}$ are essentially a transposition of E. These vectors provide insight into how each point is "truly" connected, as during the transformation of the similarity ratings, the "strength" of the adjacencies is accentuated, either increasingly, if it is a strongly connected adjacency, or decreasingly, if it is not so strongly connected.

The adjacency matrix is updated to reflect the vectors $\{y_1, ..., y_n\}$, and then it is fed into the K Means algorithm, detailed in section 2.4, in which the Matrix is partitioned in sub-matrices.

2.4 K Means Stage

K-Means algorithm takes a matrix type and clusters adjacencies based on the similarity weighting, producing a list of matrices. The algorithm continues until the weights stop changing. The initial centroids are k randomly chosen Vectors stored in a list (containing the first centroid and then growing with iterations), each iteration step the centroids are updated to the new mean. Iteration stops when the centroids stop updating.

Formally, cluster vectors $\{y_1, ..., y_n\}$ (a Matrix) using the k-means algorithm into clusters $C_1, ..., C_k$ (a list of Matrices). This involves finding a partition $C = \{C_1, ..., C_k\}$ of a set of data points $A = \{a_1, ..., a_n\}$, with the corresponding centers $c_1, ..., c_k$, which minimize the sum of the squares of distances between data points and their corresponding cluster centers.

Clusters are populated in each iteration, the algorithm chooses a vector (which is a data point listed with its adjacencies) and places it in the cluster which minimises the following distance function from that cluster's respective centroid.

$d(a, c_j) = \sum_{i=1}^{d} (a_i - c_{ji})^2$

where a is a vector of data points and c_j is the centroid vector of a cluster.

The overview of the k-means algorithm:

1. Start with an initial set of cluster centres $\{c_m : 1 \leq m \leq k\}$, The Forgy method randomly chooses k vectors from the dataset matrix and is used as the initial means.
2. Cluster all points $a \in A$ into clusters A_m by associating each $a \in A$ with the nearest cluster centre.
3. Replace cluster centres with the centroids of thus obtained clusters.
4. Repeat 2 and 3 until cluster centres (and thus also clusters) stop changing.

3 Description of Implementation

This section describes the implementation of the Spectral Clustering algorithm. More detail can be found in the code comments: for detail about the top level algorithm please see `Spectral.hs`, for detail about `Matrix` type preprocessing and decomposition please see `Matrix.hs`, and for detail about the K Means algorithm please see `KMeans.hs`.

3.1 Data Structures

The data structures, such as Matrices and Graphs, and the algorithm itself (aside from actually calculating the eigenvalues and eigenvectors) are implemented entirely from scratch using only the Haskell base library.

The most important data structure in the implementation is the `Adj` type which is a Record type used to store the weight and edge information. The Matrix representation is built up using lists of lists, containing elements of type `Adj`. `Matrix` type are lists of `Vector` type and `Vector` type is a list of `Adj` type.

In Haskell, the numeric operators such as plus, minus and multiply can be specially implemented for a custom type by implementing a set of functions for the `Num` type class. This reduces repetition of code and allows for many functions to be declared much like mathematical equations. For example, the `||+||` operator is an infix function implemented for addition of `Matrix` types, and under the hood the function maps `+` for `Adj` types in each of the matrices.

3.2 Input

The core of the program takes a list of pairs containing a score (Double type) and a label (String type). On the top level the program is set up to scrape the Yahoo Finance website. It is also possible to generate random data and run the program on this.

3.3 Output

The program will generate a `PNG` image containing a plot of the data using the `GNUPlot` Haskell API, as well as a CSV file containing the clusters separated by a blank line.

3.4 Preprocessing Stage

Much of the functions involved in the preprocessing is abstracted away from the top level by the types and the `Matrix` module's auxiliary functions.

When constructing the weighted adjacency matrix, a `Matrix` from a set of data points is constructed with the function `fromList` which converts a list of pairs of Strings and Doubles into a `Matrix`, where for each data point a adjacency list (`Vector` type) is created. The weighting for these adjacencies is determined by the function `simFn` that encodes the formula detailed in section 2.1.

When constructing the Degree matrix, first a Identity matrix is constructed using the function `identM` and then a Vector containing the row sums is constructed, and then the Identity matrix is multiplied by the Degree vector (the matrix vector product function is a infix function `||+|`), to produce the $n \times n$ Degree matrix.

When constructing the Laplacian Matrix, this is almost entirely abstracted away, it involves using matrix subtraction function on the two input matrices, the Degree matrix and the Weighted adjacency matrix.

The abstraction allows for clean function definitions and equation-like functions that mimic the algebra used to describe Spectral Clustering theory.

One shortcoming of this implementation is that whenever a function transforms a `Adj` type, the first input is the one whose Edge information is saved.

3.5 Decomposition

The stage involves temporarily transforming the `Matrix` representation into two dimensional array , and then computing the eigenvalues and eigenvectors using external software: the `LAPACK` library (which uses C under the hood). Once the eigenvectors are computed, the k eigenvectors corresponding to the smallest non-negative eigenvalues are taken and used to reconnected the `Matrix` representation.

3.6 K Means

K-Means algorithm takes a `Matrix` type and clusters points based on the similarity weighting, producing a list of matrices. The implementation involves iterating with recursive functions calls until the centroids reach a fixed point. The initial clusters are just randomly chosen. Clustering are formed by placing vectors in the clusters in which they are "nearest to" the centroid of that cluster.

Detailed comments explaining the implemenation of the K Means algorithm can be found in `KMeans.hs`.

4 The Fundamental Statistics Score & Script

The script that is used to gather the data points is a crucial component to the overall project. For a list of ASX stock codes (via a CSV file called "asx-listed-companies.csv"), the script will scrape the Yahoo finance website for a bunch of statistics relating to the code. The output of the scrape is a JSON file containing a list of JSON objects, with each of these elements corresponding to a stock. These JSON objects contain the stock code (a String) paired with the statistics for that code, now pruned and collected into a list of Doubles.

Python was used for this script because of the maturity of the scraping libraries Python has to offer.

The script builds a URL string from the stock code and the Yahoo finance URL, and retrieves the entire HTML of the webpage. The Beautiful Soup Python library is used to collect HTML objects with relevant data. The data is packaged up into a JSON object that the Haskell program will understand, and then is printed to a file.

5 Installation & Running the Algorithm

5.1 Scraping Script

The script will try and scrape every stock on the ASX, this takes about 106 mins, so if one would prefer the use the data points already scraped, please proceed to the next section 5.2

To install the requirements for the scraping script, run:

```
pip3 install -r requirements.txt
```

To run the scraping script:

```
python3 stock.py "./output.json"
```

Where the parameter is the filepath for writing the output of the scrape

5.2 Core

To install the dependencies required by the core project, run:

```
# For MacOS:
brew install ghc cabal-install gcc lapack gnuplot

# For Linux:
sudo apt install ghc cabal gcc lapack gnuplot
```

Change into the Haskell project directory

```
cd hs-stock
```

Update dependencies (you should only have to do this once):

```
cabal update
```

To enter into an interactive session with GHC, this will start a prompt in which functions of the project can be run from:

```
cabal repl
```

And to run the main function of the project, enter:

```
main
```

Note: if you get a failure to parse JSON, the script has misprinted a comma. I recommend using a "jsonformatter" online to fix this.

The prompt will as for arguments in the format:

```
<k> <sig> "<filepath-to-data>"
```

Where `k` and `sig` should be replaced by the number of clusters desired and the neighbourhood size. For example:

```
10 0.25 "../saved-output-1.json"
```

Note: the pre-scraped data points are stored in the file "saved-output-1.json".

The plots ("original-data-plot.png" and "clustered-data-plot") and CSV ("results.csv") file can be found in this directory after main has finished running.

It is also possible to run clustering on randomly generated pseudo-clusters, when in REPL enter:

```
spectralRand
```

Please contact me if there's any issues running this.

6 Reflections

6.1 Experience with implementation

Implementing the Spectral Clustering algorithm has been quite difficult and perhaps the Haskell language was not the best fit. The main reason Haskell was chosen was because of appealing mathematical elegance, however I am still a novice at the language and this project has been an incredible learning experience. And Haskell's type safety is really good for mission critical applications, but perhaps not this project as it made development painful sometimes. One positive point about using Haskell was the incredible abstraction of logic it provides, which is a joy to program when you get it (but when you don't it is painful...).

Throughout development there was a lot of refactoring and a lot of functions that were scrapped, much of the time was spent getting the foundations set up and this meant there was not so much time left for extras at the end of the project. In retrospect I should of clearly defined the inputs and outputs first, and then narrowed down a data model before doing anything other coding.

A lot of time was spent just trying to understand the Spectral Clustering algorithm, as this topic was only covered in later lectures, and kept finding myself thinking I'd understood it but then later realising that my understanding was not quite correct. And as a result, a lot of time was spent testing, which perhaps not the most efficient methods of testing used.

Another tricky task was trying to preserve the geometry of the matrix while operating on it.

6.2 Shortcomings, fine tuning and extensions

One optimisation that is fairly achievable would be to switch from using `List` types to using `Vector` types from the Data.Vector library. This library is essentially a safe wrapper around efficient (mutable and immutable) arrays, with a powerful loop optimisation framework.

The K Means algorithm can be improved with a more intelligent pick of the initial centroids, as in the implementation the initial centroids are randomly chosen.

The types used can be improved, in the implementation they are Record types, however the Haskell ecosystem contains many fancy types that could make the code cleaner and more reusable.

In the end, the implementation did not produce very good clusters from the stock data. Throughout development clustering was working (with random data) until I integrated the stock data, this forced me to change my representation of data. So, in order for clustering to work as intended more time needed to be spent on getting a nice data set. Therefore, paying for API access would definitely improve clustering, or even using data from years past would improve it.

In conclusion, this major project has been challenging and rewarding. Although I did not come up with a 100% working solution, this is due to the input data set, not the implementation itself. The implementation is solid thanks to Haskell's type safety and it was tested throughout development with random data points (which formed rough clusters). I have learned a incredible amount: my Haskell programming proficiency has vastly improved, my Linear algebra knowledge has been fortified and I've learnt a great deal about how to properly implement a project.

References

[1] URL: https://cse.buffalo.edu/~jcorso/t/CSE555/files/clustering_CSE555_guestlecture.pdf.

[2] Aleks Ignjatovic. *Lecture notes in Advanced Algorithms Course*. Dec. 2020.

[3] Ulrike Luxburg. "A Tutorial on Spectral Clustering". In: *Statistics and Computing* 17.4 (Dec. 2007), pp. 395–416. ISSN: 0960-3174. DOI: 10.1007/s11222-007-9033-z. URL: https://doi.org/10.1007/s11222-007-9033-z.