University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Visualisation of Mobile App Usage

Matthew James O'Hare

April 06, 2020

Abstract

Typical large-scale datasets now consist of dimensions greater than 3 or 4 which can prove troublesome for both analysis and visualisation. To solve this, dimensionality reduction algorithms were created and have developed over the years to incorporate more advanced techniques. This project sought to visualise iOS device data obtained through an app created by University of Glasgow School of Computing Science, using some of these different algorithms. The aim was to evaluate how well these algorithms visualised this data and the cost of it. With the largest dataset sample consisting of over 10,000 entries and 45,000 dimensions, it was found that more advanced techniques prevailed, both in terms of visual layout, runtimes and memory usage. The latter two are what greatly limit the practicality of older methods when dealing with large high-dimensional datasets, although their produced layouts are sound.

Acknowledgements

I'd like to thank Professor Matthew Chalmers as the project supervisor for his continuous support, feedback and guidance throughout this project.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Matthew James O'Hare Date: 06 April 2020

Contents

1 Introduction

1.1 Motivation

It is predicted that around 2.87 billion users across the globe own a mobile device (Butt 2020). It is therefore expected that usage patterns will vary across every user and with over 2.2 million iOS applications alone, analysis of these patterns can become problematic. Previous research of app usage has involved statistical methods however this limits the understanding of the data which can be achieved.

An alternative to statistical methods which can improve this understanding is the creation of a meaningful visualisation. Since this data may span many columns and therefore dimensions, this would require high-dimensional data to be visualised in a low-dimensional space while approximating high-dimensional distances between objects. This low-dimensional space is typically 2 or 3 dimensions although 2 dimensions is commonly used due to its ease and ability for data exploration and general overview compared to 3 dimensions which may not be as beneficial. This deeper understanding mentioned may include trends, identification of outliers and the overall structure of the data which is otherwise troublesome to identify with statistical methods.

As common visualisation techniques do not scale well with dimensionality a new class of algorithms has been developed; dimensionality reduction algorithms. With the rise of machine learning, advanced techniques have improved the efficiency and overall ability of these algorithms when dealing with high-dimensional data.

University of Glasgow School of Computing Science, in separate projects, has created three successful force-directed graph layout algorithms (Chalmers; Morrison et al. 2002; Morrison and Chalmers 2003), a subset within dimensionality reduction. They have also obtained app usage data from over 400,000 users through a series of developed iOS logging applications. As the data was only analysed statistically, it was suggested that it may be beneficial for researchers if it was used to create meaningful visualisations through dimensionality reduction algorithms.

1.2 Project Description

This project seeks to visualise iOS app usage data in a meaningful way. In doing so, a comparison will be carried out between dimensionality reduction algorithms in Python. This will include the implementation of these algorithms, if there is no existing implementation available, as well as an evaluation using specific performance metrics in order to create a fair comparison. Simply, the aim here is to compare, in an unbiased way, different visualisation algorithms applied to the same data.

1.3 Summary

This chapter introduced the project including its aims and the motivation behind it. The remaining chapters will cover the following:

- **Chapter 2** provides an overview of the dataset, algorithms and the performance metrics which will be used.
- **Chapter 3** outlines the requirements of the project and the technologies used to satisfy them.
- **Chapter 4** explains the process of how the project's design requirements were met and any challenges that were overcome.
- **Chapter 5** presents a fair comparison of the mentioned dimensionality reduction algorithms using specified performance metrics.
- **Chapter 6** describes the outcome of the evaluation, reflection on the project and proposed future work.

2 Background

Many techniques exist for the visualisation of high-dimensional data, each carrying their own advantages and disadvantages. This section will discuss some of these approaches as well as the background of dimensionality reduction algorithms and the problem they seek to solve.

2.1 Curse of Dimensionality

The "curse of dimensionality" is a very important problem within data science. It is a phenomenon that occurs when classifying, organising and analysing high-dimensional data that does not occur in low-dimensional spaces, specifically the issue of data sparsity and "closeness" of data. (DeepAI 2019a). As more dimensions are added, the volume of the vector space increases exponentially, making generalisation on high dimensions difficult. Because of this, many statistical methods lack power when applied to high-dimensional data (Nguyen and Holmes 2019). Even if the number of data points is large, they remain sparsely embedded in a voluminous high-dimensional space that is practically impossible to explore exhaustively. By reducing the dimensionality of the data, the aim is to improve exploration and in turn analysis but at the cost of some form of approximation. This cost is required since we cannot remove the impact of this troublesome phenomenon, instead we can only attempt to alleviate it as much as possible. As dimensions are reduced, it becomes more common that objects are relatively close in the space, regardless if they were originally clearly separated in high-dimensional space. As a result distance metrics struggle to discriminate between objects that are slightly different and those that are very different. Therefore, careful planning and implementation is required to find a balance.

2.2 Dimensionality Reduction

The objective of dimensionality reduction algorithms is to find an embedding of high-dimensional data in a lower dimensional space that retains important characteristics of that data. These important characteristics may include preserving high-dimensional relationships between data points.

Dimensionality simply refers to the number of independent features in a dataset where features are defined as the number of input variables or independent columns (EliteDataScience 2019). However many algorithms do not take this independence into account therefore the number of columns simply becomes the number of dimensions. There are two main categories of dimensionality reduction techniques, feature selection and feature projection. Feature selection techniques are where only the most important or descriptive features are kept and the remaining irrelevant or redundant features are discarded (Vlachos 2010). On the other hand, feature projection techniques transform data in the high-dimensional space to a space of fewer dimensions. Early techniques, like PCA (Pearson 1901), create linear combinations of original features with the aim of finding new features that retain dataset structure and its variance as closely as possible. However as more sophisticated techniques have been developed, they have struggled to be categorised in either of the two outlined. Spring Models, t-SNE and UMAP are just a few examples of this. Although they all transform data onto lower dimensions, similar to feature

projection techniques, all three of these algorithms are non-linear which differs them from these older methods:

- Spring Models are a concept where forces are applied between all pairs of nodes which pull distant but similar nodes towards each other, and push close but dissimilar ones apart (Eades 1984; Chalmers; Kobourov 2012).
- t-SNE computes the probability that pairs of data points in the high-dimensional space are related and then creates a low-dimensional embedding that produces a similar distribution (van der Maaten and Hinton 2008).
- UMAP is similar to t-SNE with the exception of a few assumptions, mainly based on the Riemannian manifold and metric (McInnes et al. 2018).

These non-linear transformation techniques can also be used on data which is amenable to linear algorithms. This is beneficial since it removes the need to make an assumption about the linearity of the data in question. They are based on the manifold hypothesis which says that in a high-dimensional structure, most relevant information is concentrated in a small number of low-dimensional manifolds. These three algorithms, Spring Models, t-SNE and UMAP, will be the focus of this project.

2.3 Previous Student Projects

In 2019, a Python implementation was created of Chalmers' 1996, Pivot Layout and Hybrid Layout spring model algorithms (Cattermole 2019). These were adapted from previous JavaScript implementations from 2017 and 2018 respectively (Bartasius 2017; Boonsarngsuk 2018). The most recent implementation was used a starting point for this project to implement the Spring Models mentioned. However, due to the different aims of those projects and datasets used, the results of these projects will not be directly compared to this one.

2.4 Related Work

Background research for this project involved looking at previous studies on the visualisation of app usage data or visualisation for app analytics. Previous studies were aimed at the latter topic and consisted of Android users, in particular, Böhmer et al. (2011); Welke et al. (2016), while a large scale study by Morrison et al. (2018) consisted of iOS users, a first of its kind.

Böhmer et al. (2011) presents a study concentrating on day to day app usage of over 4100 users, including average app launches per session. As well as this, individual and categorical app usage by hour of day and the transition probabilities of a user from app category to app category were found showing the average session lasts less than a minute with utilities and news apps most popular in the morning and games at night but communication apps dominate throughout most of the day. This statistical data was then visualised in chart form to provide further analysis. A larger study, Welke et al. (2016) sought to prove that identification of a user is possible based on their app usage. It was found that 99.4% of app signatures were unique among the 60 most popular apps while 99.67% of app signatures were unique among the 500 most popular. Using Hamming distance as a metric of distance between users' app signature, this statistical data was visualised through simple charts to show the variation in distances between all users. Morrison et al. (2018) sought to reproduce the two mentioned studies but using iOS users as opposed to Android users, combining the features of both together with similar results and visual charts.

These three studies all use different app usage datasets and the first two concentrate on different features of the data with the third combining them, however they all share the same method of analytics; statistical data and simple visualisations. This is a traditional method of analytics that has been the norm for some time now. No previous study has attempted to use dimensionality

reduction algorithms, or similar visual methods, to visualise and analyse app usage data in an alternative way to these traditional methods which is what this project seeks to change.

2.5 Algorithms

This section will detail the force-directed graph drawing and dimensionality reduction algorithms that will be compared and analysed in this project. These include three spring model algorithms, a basic dimensionality reduction algorithm and two more complex ones.

2.5.1 Random Projection

Random projection is a mathematical method simply used to reduce the dimensionality of a dataset by projecting the data points efficiently to a smaller number of dimensions while preserving the original relative distance (Nabil 2017). The key idea of random mapping originates from the Johnson-Lindenstrauss Lemma which states if points in vector space are projected onto a randomly selected subspace of suitably high dimensions, then the distances between the points are approximately preserved (Bingham and Mannila 2001). Since it is computationally efficient and accurate enough for preserving basic structure in the data, it was used as a baseline for comparing more complex algorithms to.

2.5.2 Spring Models

Force-directed placement techniques, also known as Spring Models, belong to a class of algorithms called multidimensional scaling (MDS) which is a form of non-linear dimensionality reduction. The goal is to map a data set onto a smaller number of dimensions while creating a visual representation that preserves relationships within that set of objects (Morrison et al. 2002). Simply, Spring Models determine where a high-dimensional point is laid out in low-dimensional space based on inter-object similarities. This similarity is more commonly a calculated dissimilarity in high-dimensional space typically using Euclidean distance, which is then approximated in low-dimensional space. There are three algorithms that will be concentrated on during this section; Chalmers' 1996, Hybrid Layout and Pivot Layout algorithms.

Chalmers' 1996 Algorithm

In 1996, Chalmers introduced a new method to force-directed graph drawing. This method significantly improved upon previous approaches like the original Brute Force implementation created by Eades (1984), reducing complexity from $O(N^3)$ to $O(N^2)$ (Chalmers). These brute force implementations typically calculate a force between each pair of nodes and as a result there are many calculated forces, typically nodes that are far apart, that simply do not make a difference to the overall layout. It was this issue which Chalmers sought to improve and the approach is outlined below.

Two sets are created, a neighbour set storing values in order of distance in high-dimensional space being carried over between iterations and a sample set consisting of a random subset of objects, reconstructed before each iteration. Both have a constant maximum size. During each iteration, the distance between each node in the sample set would be compared with the neighbour set. If the calculated distance is less than *maxDist*, the current maximum distance to any node in the neighbour set, then the current node is placed in the appropriate ordered position in the neighbour set. This might remove the most distant neighbour in the process. In this manner, a neighbour set becomes more representative of the closest nodes as iterations progress (Morrison et al. 2002). As a result, each node builds up a neighbour set which is likely to contain only the nodes that have the lowest high-dimensional distance from it.

This method meant that the algorithm spends less time calculating small, insignificant forces between nodes that are very far apart and do not alter the overall layout, allowing a similar quality layout to be achieved with a far smaller computational cost, $O(N^2)$.

Hybrid Layout Algorithm

In 2002, Morrison, Ross and Chalmers published a new approach which improved on the complexity of Chalmers' 1996 algorithm, reducing it from $O(N^2)$ to $O(N\sqrt{N})$ (Morrison et al. 2002). This was achieved using three stages to the algorithm. The first stage involved taking a random sample of $\sqrt{N}$ of the original dataset and using this sample with Chalmers 1996' algorithm - running in $O(N)$. To complete the layout with the remaining $N - \sqrt{N}$ data points, a modified version of Brodbeck and Giradin's original interpolation strategy (Morrison et al. 2002) is used on each of the $N - \sqrt{N}$ objects and compared to each $\sqrt{N}$ sample. This stage is the dominant factor, bringing the layout to a complexity of $O(N\sqrt{N})$.

For each data point, the node in the sample layout with the smallest high-dimensional distance from it is found, known as the parent. Then a circle of radius r, where r is proportional to the high-dimensional distance between the data point and its parent, is defined and the quadrant in which the lowest difference between actual layout distances and desired distances, is located. A binary search is performed on this quadrant to find the best location of the current data point. Once this node has been placed, a random sample of the original subset is selected and the sum of the forces is applied to the new node. This process is repeated a constant number of times to refine the interpolation. The final stage involves a constant number of iterations of Chalmers' 1996 algorithm being run on the whole layout to refine placement. As this model is linear per iteration, the overall complexity remains $O(N\sqrt{N})$, creating a solid basis for future implementations.

Pivot Layout Algorithm

One year after the Hybrid Layout approach was published, an improvement was proposed which used pivots with pre-calculated distances to the sample set to decrease the time complexity of the interpolation stage (Morrison and Chalmers 2003).

This is achieved by firstly creating a random sample from the initial $\sqrt{N}$ sample layout to act as pivots and the high-dimensional distance between each pivot to all other sample nodes is pre-calculated. Each pivot is assigned a fixed number of buckets, each representing different ranges of distance from the pivot. Each non-pivot node within the initial sample is assigned to a bucket based on the high-dimensional distances pre-calculated. During interpolation of a new node, the distance to each pivot is calculated and the appropriate bucket for this new node is determined. Finally, the contents of this bucket is searched for the overall nearest neighbour or the node with the smallest high-dimensional distance to the new node, which becomes the parent. This approach leads to an overall time complexity of $O(N^{5/4})$ which is an improvement over the Hybrid Layout's $O(N\sqrt{N})$.

2.5.3 t-SNE

t-Distributed Stochastic Neighbour Embedding, more commonly referred to as t-SNE is a non-linear dimension reduction technique particularly well suited for the visualisation of high-dimensional datasets (van der Maaten and Hinton 2008). Initially the algorithm had a computational complexity of $O(N^2)$ however an optimisation using the Barnes-Hut approximation during the gradient calculation stage results in an improved performance, albeit slightly less accurate, of $O(N \log N)$. Firstly the algorithm calculates the probability of similarity between pairwise points in the original high-dimensional space, and in an initial embedding in the corresponding low-dimensional space. Next, the algorithm attempts to move objects in the low-dimensional space, so as to minimise the difference between the second of these probabilities to find the

best representation of the data points in low-dimensional space. This minimisation is of the Kullback-Leibler divergence, which is a measure of how one probability distribution diverges from a second. This results in a mapping to low-dimensional space of the multi-dimensional data where patterns can be visually identified.

2.5.4 UMAP

Uniform Manifold Approximation and Projection (UMAP) is another dimension reduction technique that can be used for visualisation similarly to t-SNE, but also for general nonlinear dimension reduction (McInnes et al. 2018). The complexity for UMAP has not been formally established although the authors of the original paper suggest an empirical complexity of $O(N^{1.14})$. The algorithm is founded on three assumptions about the data:

- The data is uniformly distributed on Riemannian manifold;
- The Riemannian metric is locally constant (or can be approximated as such);
- The manifold is locally connected

From these assumptions it is possible to model the manifold with a fuzzy topological structure (Zadeh 1965). An embedding is found by searching for a low-dimensional layout of the data and optimising this layout so that it has closest possible equivalent fuzzy topological structure, as measured by cross-entropy (UMAP-Learn 2018). Once this optimised embedding is found, it is visualised in low-dimensional space.

2.6 Dataset

As per the project description, past research in the School of Computing Science has focused on logging in detail the use and execution of applications for mobile phones. As a result, usage has been gained from thousands of users that has yet to be explored and understood. In this section, the dataset used in the project is presented, both the dataset itself and how it was harvested.

2.6.1 App Usage

This dataset was obtained via AppTracker, an app created for jailbroken iOS devices created by the University of Glasgow School of Computing Science. The app was publicly available on an unofficial third-party repository until 2018 and as of September 2017, it was downloaded over 40,000 times. The app consisted of a background logging framework that captures information on device use, and a foreground UI that displays charts and statistics on app use durations. AppTracker recorded timestamped logs of every time an app was opened or closed on the device along with tracking every time the device was locked or unlocked. This allowed the time an app has been active in the foreground to be determined. It is important to note that AppTracker does not record information on activity within individual apps or activity over networks (Morrison et al. 2018). During this app's lifetime, over 28 million app launch events were recorded. For the purpose of this project, the dataset used has been gathered from 10,368 users with 45,788 unique apps used. The original data was anonymised to restrict identification of users, either through app usage (Welke et al. 2016) or unique information such as device identifier, and split into four files as follows:

- User_launch_counts_appAnon.csv
- Genres.csv
- Duration_seen.csv
- App_genres.csv

User Launch Counts
This file contained 3 columns with 289,143 rows. The first column was *userID*, the second column was the number of launches of an app with the third column being the *appID* being referenced.

Genres
This file contained 2 columns and 25 rows. The first column was the *genreID* and the second column was the genre name. A genre is category in which an app falls under, examples may include social networking, news or productivity.

Duration Seen
This file contained 2 columns and 10,368 rows (representing one row per user). The first column was the *userID* with the second column being the duration, in seconds, for which their activity was tracked.

App Genres
This file contained 4 columns and 45,788 rows (representing one row per app). The first column was the *appID* with the second column being the *genreID*, the third was *apple* which contained a binary value depicting whether the app was a built in Apple application. The fourth column was *manualGenreAssignment*. This also contained a binary value depicting whether the *genreID* was assigned manually. The reason for this manual assignment could be for a multitude of reasons, however for most cases, the app changed category or belongs in multiple categories, and so one was manually assigned by Morrison et al.

These four files and the resultant dataset created from them, discussed in Section 4, formed the basis of this project entire and as such are referenced throughout it.

2.7 Performance Metrics

In order to compare these algorithms, a common set of metrics had to be used. This section will explain these metrics as they will be used in the evaluation.

2.7.1 Runtime

As these algorithms would be used with large high-dimensional datasets, the time to execute an algorithm to create layouts with this data is important. This was the first metric. Runtime is defined as the period of time for the algorithm to complete execution. This excludes the timed used for rendering of layouts, or any other calculations outwith the core layout algorithm. Since each algorithm is different, in terms of logic and therefore algorithmic complexity, it was expected that each algorithm would result in different runtimes.

2.7.2 Memory Usage

Another important factor to take into account when using these algorithms with large high-dimensional datasets is the memory required for the algorithm to run and create a layout. As each algorithm uses progressively more complex techniques, it is possible that this may require more memory. This memory usage is defined as the RAM required to run the algorithm, again excluding any usage by rendering tools or other calculations outside the algorithm.

These conditions allow for a direct comparison to be made between algorithms with respect to their runtimes and memory usage.

2.7.3 Visual Layout

The last and arguably most important metric is the quality of the visualisation produced by the algorithm. Unlike the other two metrics where a value can be assigned to each algorithm for it, this metric is slightly more subjective. In particular, this metric should take into account:

- The accuracy of high-dimensional clusters on the low-dimensional layout
- The overall position and accuracy of data points, including distance between points and clusters

This metric will be subjectively measured using the above factors and will therefore assist in comparisons between visualisations of the mentioned algorithms.

2.8 Summary

This chapter discussed the background of dimensionality reduction algorithms and the problems they attempt to overcome when visualising high-dimensional data. Various techniques were introduced, including the six algorithms that will be compared in this project: Chalmers' 1996, Hybrid Layout, Pivot Layout, Random Projection, t-SNE and UMAP. These were outlined and the dataset that they will be used with - App Usage - was introduced. Lastly the performance metrics: runtime, memory usage and visual layout which will be used as part of the evaluation to compare these algorithms were discussed.

3 Design

This section outlines the design decisions made in the beginning of the project to facilitate the project requirements. This includes the requirements themselves and the technologies that were considered to satisfy them.

3.1 Requirements

As per the project brief, the initial requirements were left open to interpretation. As a result, a project scope was required.

3.1.1 Pre-processing

One of the most important aspects of large scale analysis/visualisation projects is preparing and examining the data so that it can be visualised in a meaningful way. This involves data cleaning as the first step where missing values are filled in, noisy data is smoothed and outliers are identified and removed if necessary (Nguyen and Holmes 2019). The next step is data transformation where data is normalised and aggregated to produce an appropriate output which can be used for visualisation. Therefore the data, outlined in Section 2.6, must undergo this process.

3.1.2 Metrics

Once data has been pre-processed, appropriate distance metrics must be identified and compared in order to measure and best model the data.

3.1.3 Clustering

It is important to identify clusters, both in high-dimensional and low-dimensional space, within the data so they can be visualised on the layouts and aid in the comparisons between algorithms and their visualisations.

3.1.4 Algorithms

Existing spring model code, in Python, was provided which served as a starting point. This existing code contained four force directed graph layout algorithms:

- Brute Force algorithm
- Chalmers' 1996 algorithm
- Hybrid Layout algorithm
- Pivot Layout algorithm

The initial idea was to use these algorithms with the pre-processed data outlined above. As the project progressed and a greater understanding was achieved, this idea was extended and refined to the following algorithms:

- Chalmers' 1996 algorithm

- Hybrid Layout algorithm
- Pivot Layout algorithm
- Random Projection
- t-distributed stochastic neighbor embedding (t-SNE)
- Uniform Manifold Approximation and Projection (UMAP)

As a result, Random Projection, t-SNE and UMAP would have to be implemented while using the existing code to provide implementation for Chalmers' 1996, Hybrid Layout and Pivot Layout algorithms. The appropriate distance metric, discussed above, must be used in conjunction with these algorithms to optimise these implementations.

3.1.5 Visualisation

Since the purpose of the algorithms and the project is visualisation, functionality to display and save these layouts is required. This includes colouring nodes to identify clusters and implementing interactive layouts with specific annotations attached to each node to allow for analysis of data points and clusters.

3.1.6 Evaluation

A thorough evaluation must be carried out to compare the layouts using appropriate metrics, discussed in Section 2.7.

- Runtime: The time taken for the algorithm to run must be measured, using Python's `time.perf_counter()` to ensure consistency.
- Memory usage: The RAM usage of the algorithm must be measured , using Python's `sys.getsizeof()` to ensure accuracy.
- Visual: The visualisation itself must be analysed using the factors discussed in Section 2.7.3.

The first two metrics should be compared in graphical form with the third being compared in visual and analytical form.

These requirements encompassed the scope of the project and were carefully considered when choosing the most appropriate technologies for the project.

3.2 Technologies

This section will discuss the technologies considered when planning the implementation of the project. It will consist of a description of each technology considered and the purpose it serves to the project.

3.2.1 Python 3.7

This was a requirement listed in the original project description, as mentioned in Section 1.2, specifying that the project would be written using the Python programming language. Python 3.7 was chosen as it provides the most up to date features over previous versions (Python Software Foundation 2018).

3.2.2 SciPy

SciPy is a Python-based ecosystem consisting of a collection of open-source software for mathematics, science and engineering (SciPy 2020). Some of the core packages include NumPy, SciPy library, Matplotlib and pandas. The SciPy library provides many efficient modules, in particular *SciPy.spatial* which among other things, provides utilities for distance computations using various metrics. This covers the specifics of requirement 2, making it a suitable choice for the project.

3.2.3 Pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool built on top of the Python programming language (pandas 2020b). With critical code paths written in Cython or C, this makes use of the speed of the C programming language and thus is optimised for high performance. With many inbuilt features such as the efficient DataFrame object with integrated indexing, high performance merging and joining of data sets and tools for reading and writing data between in-memory data structures and formats, for example, CSV, this library provides all of the tools required to load, manipulate and output data efficiently. As this project was dealing with large datasets, careful pre-processing was required and pandas was the most appropriate library to facilitate this.

3.2.4 NumPy

NumPy is a fundamental package for scientific computing in Python providing high-performance multidimensional array objects, and tools for working with these arrays efficiently (NumPy 2020). Implementation of the algorithms must be efficient as they are working with large datasets and these objects provide superior speeds and memory usage, due to also being mostly written in C, than basic Python lists. Therefore, NumPy provides all the tools necessary to implement efficient algorithms.

3.2.5 Matplotlib

Matplotlib is a comprehensive library for creating and saving static, animated, and interactive visualisations in Python (Matplotlib 2020). This allows custom colours, annotations and hover events within layouts which covers the specifics of requirement 5, justifying the decision to use it throughout the project.

3.2.6 Seaborn

Seaborn is a Python data visualisation library based on Matplotlib providing a high-level interface for drawing attractive and statistical graphs (Seaborn 2020). This will be used to satisfy requirement 5; to visualise comparisons between algorithms and layouts.

3.2.7 Scikit-learn

Scikit-learn provides simple and efficient tools for predictive data analysis built upon NumPy, SciPy and Matplotlib (scikit learn 2020b). These tools include classification, regression, clustering and dimensionality reduction. It is the latter two tools which are important to this project as they allow clusters to be found within the data, satisfying requirement 3, and dimensionality reduction algorithms, in particular t-SNE and Random Projection, to be applied to the data, satisfying requirement 4.

3.3 Summary

This section has outlined the specific requirements of the project including pre-processing, metrics, clustering, algorithms, visualisations and evaluation. The technologies used to satisfy these requirements: Python 3.7, SciPy, pandas, NumPy, Matplotlib, Seaborn and Scikit-learn were also discussed and how they related to the project.

4 Implementation

Throughout the project, a step by step process was used and a number of challenges and decisions to be made arose which contributed massively to the overall quality of the project. This section explains these in detail and the implementation cycle itself.

4.1 Pre-processing

In the beginning, careful planning and pre-processing of the four files, discussed in Section 2.6, was required to create an appropriate dataset for visualisation. The four original files had to be imported into an appropriate data structure which would allow for fast and efficient manipulation and indexing of its contents. As mentioned Section 3.2.3, this was chosen to be a pandas DataFrame.

4.1.1 Feature Extraction

Starting with "User_launch_counts_appAnon.csv" as the first file for import and analysis, the id's of all users and apps were first obtained using a simple in-built `unique()` pandas function. Using `value_counts()` confirmed that there were 10,368 users and 45,788 apps across the data.

	userID	launches	apps_used	average_launches	duration_seen
0	2	614	49	12.530612	138689
1	4	65	11	5.909091	481495
2	5	22	7	3.142857	4322898
3	6	1	1	1.000000	0
4	7	19	5	3.800000	5731
...	...	...	...	...	...
10363	16186	348	22	15.818182	188819
10364	16189	360	25	14.400000	800404
10365	16191	66	19	3.473684	3170203
10366	16193	2	2	1.000000	56
10367	16194	68198	235	290.204255	69221062

10368 rows × 5 columns

Figure 4.1: *DataFrame created to represent each user and their general usage across all apps.*

Several dictionaries had to be created to store different features of the data which were combined and mapped to a new DataFrame. The first dictionary used *userID* as the key with the list of appIDs associated with that *userID* as the corresponding value. The second dictionary similarly used the *userID* as the key with the corresponding list of counts of launches for each app used as the value. These two dictionaries were then combined with the "duration_seen_seconds"

column in "Duration_seen.csv" to create an informative DataFrame about each user. The new DataFrame, as shown in Figure 4.1, lists useful information associated with each user. This includes, total number of launches, total number of apps used, average number of launches and the total duration of time that data was recorded for that user.

A similar process was repeated with the focus on apps, instead of users. The next dictionary contained each *appID* as the key with a list of each *userID* that was associated with that *appID* as the corresponding value. The fourth and final dictionary similarly contained each *appID* as the key and a list of the counts of each launch for that app by every user that used the app. Again, combining these two dictionaries together allowed the sum, the length and the average count of each app to be calculated. These values were then aggregated together to create another useful DataFrame focusing on apps and this can be seen in Appendix A.1.

4.1.2 Removing Inactive Users

With 10,368 users in the data, the length of time that we have seen each user dramatically ranges from 0 to >10,000,000 seconds (this can be seen in the "duration_seen column" in Figure 4.1). As a result, problems may arise such as where less active users may not have enough activity to be similar to any other user, creating outliers in the visualisations. An example of this was where one user had AppTracker downloaded for around 50 days yet had only 22 total launches between 7 different apps in that time. This is clearly outlier behaviour which may limit the usefulness and accuracy of our results if not removed. Therefore the decision was made to filter out these inactive users. Only 55.6% of users (5768) had used AppTracker for longer than one day meaning removing 4600 users for less than one day of usage would not be suitable. A a result, more appropriate conditions were required where users were excluded if they fell into one of these categories:

- Users seen for less than five minutes.
- Users seen for less than one day, using less than one app per hour.
- Users seen for more than one day, using less than one app per day.

This brought the count of active users to 7504, excluding 2864 inactive users in the process.

4.1.3 Creating Activity DataFrame

In order to calculate the distance or similarity between users, covered in the next stage, careful consideration was carried out to use the most appropriate structure and format to facilitate this. Two structures were suggested; a dictionary and a DataFrame. A DataFrame can be thought of as a dictionary-like container for Series objects (pandas 2020a) where a Series is a one-dimensional array with axis labels (pandas 2020c). These axis labels are a row index.

The requirement was to have each user and the list of apps they had used combined with the number of launches of each. The suggested dictionary format would consist of a nested dictionary of length 7504 each representing a user:

```
{2: {1 : launches, ... }, ... , 16194: {1:launches, ... }}
```

The value to each key would be a nested dictionary containing a key for every app used by that user with the corresponding launch count as the value. Apps not used would not be stored. The other suggestion was a sparse DataFrame with each app given a column and each active user given a row. This would result in a 7504 x 45788 DataFrame. For each user, and each app they used, the corresponding column for that app in the DataFrame would be given the value of the corresponding launch count. Every app not used by a user would be given a 0. An example of this format can be seen in Table 4.1.

Table 4.1: *Suggested format of a DataFrame consisting of a row for each user and a column for each app.*

userID	**appID**		
	1	**...**	**45788**
2	launches	...	launches
...	...	...	...
16194	launches	...	launches

Both data structures are suited to different situations however it was decided that a DataFrame would be the most appropriate structure here given the size of the data that would be handled. A dictionary is extremely efficient for small datasets however as the number of columns increase, DataFrame performance and usability outweigh that of a dictionary. Furthermore, a DataFrame is preferred to a dictionary for data analysis due to its visual ease, flexibility and ability to perform row and column calculations and manipulation.

However it was realised that due to the large difference in counts for some apps, for example, the most and least popular apps, results were skewed and disproportionate with one another. To overcome this, it was decided that normalisation should be carried out across the whole DataFrame. This involved the count for each specific app per user being divided by the average count of that app over the entire dataset. As each count was calculated with respect to its average, this created a far more proportionate output.

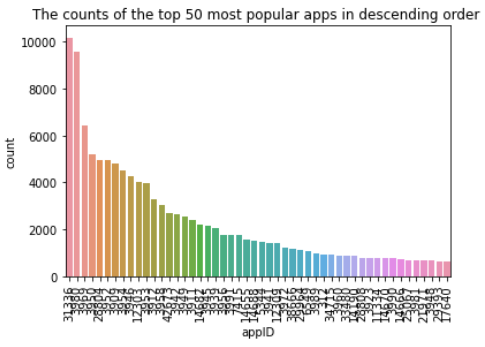
Although the original unnormalised DataFrame consisting of just the corresponding launch count for each app and user was not useful to take forward, it allowed further investigation into the apps themselves. This led to the retrieval of the total counts for each app showing the number of users that were recorded to have used each app. The DataFrame created is shown below in Figure 4.2a.

It was discovered that the most used app was a utilities app, the second being springboard (iOS home screen) with the third most used app being a utilities app also. Due to the anonymity of the data and how the original dataset was structured, the specific app name is unknown. Figure 4.2a can be visualised to provide a greater understanding, Figure 4.2b, which shows the gradual decrease in popularity between the top 50 apps.

	appID	count
0	31336	10173
1	3980	9586
2	3969	6452
3	3950	5207
4	28804	4952
...	...	...
45783	25427	1
45784	25426	1
45785	25424	1
45786	9468	1
45787	45788	1

45788 rows × 2 columns

(a) *DataFrame showing the top 5 and bottom 5 apps by their usage.*



(b) *Top 50 apps ordered by usage.*

Figure 4.2: *The most and least popular apps, in descending order.*

Following on, analysis could be conducted to find the proportion of users and apps. The main points identified from this were:

- The top 10 apps account for 20.4% of rows in the original dataset
- The top 50 apps account for 40.6% of rows in the original dataset
- 63.7% of apps (29,167) have only been used by 1 user
- 99.0% of apps (45,349) have been used by less than 50 users
- This means there are only 1% of apps (439) that have been used by more than 50 users.

It was expected that because of the vast amount of apps, the range of the number of users per app would also be large and this was proven. This illustrates the sparseness of the entire dataset, further proving why careful planning was required.

4.2 Experiment 1 - Choosing a Metric

A metric, with respect to dimensionality reduction algorithms, is a function that defines distance between each pair of elements in a set. There are many metrics to choose from but finding the most appropriate is required. Each metric calculates distances between two points in different ways, which in some cases, can result in completely different outputs. Therefore a variety of different metrics were selected initially and compared. The comparisons were all done offline and standalone; not with any other metrics or dimensionality reduction algorithms. The purpose was to rule out, based on outputs only, inappropriate metrics. A metric would be deemed inappropriate if the distance between two points did not reflect that in the dataset, taking into account other computed distances also. For example, two comparisons with 4 users may give a similar score yet only one set of users are visually similar, based on their usage. The metrics compared were:

- Euclidean Distance
  - The "ordinary" straight-line distance between two points
- Standardised Euclidean Distance
  - Euclidean distance where each value is divided by its variance
- Hamming Distance
  - The number of edits to change one string into another
- Cosine Similarity
  - The cosine angle between two non-zero vectors
- Jaccard Similarity
  - Compares two sets to see which members are shared and which are distinct
- Manhattan Distance
  - The distance between two points measured along axes at right angles

4.2.1 Process

Extensive analysis was performed to find the most appropriate. This analysis was in the form of a mini experiment with the process as follows. Four random users (User 2, 4, 7, 10) and their usage of the top 80 apps were compared and the estimated similarity between different combinations of users was recorded. Since each metric uses a different scale, defining these estimated similarities in number format would be unfair and so a qualitative approach was taken.

- User 2 and 4 = Least similar
- User 2 and 7 = Not very similar (although more similar than 2 and 10)
- User 2 and 10 = Not very similar
- User 4 and 7 = Quite similar
- User 7 and 10 = Most similar

With the exception of Standardised Euclidean, the metrics listed above were then used on the sample data and the results noted and compared. The reason for the absence of Standardised Euclidean was that it simply gives a normalised form of Euclidean distance therefore it was more appropriate to compare Euclidean on its own. Table 4.2 shows the results of these comparisons between users and the results explained below.

Table 4.2: *Similarity between users using different distance metrics*

Metric	Users				
	2 & 4	**2 & 7**	**2 & 10**	**4 & 7**	**7 & 10**
Euclidean	2.97	2.23	2.56	1.55	0.37
Hamming	22	18	18	13	4
Manhattan	6.83	4.85	5.29	2.18	0.44
Cosine	0.938	0.133	0.941	0.628	0.627
Jaccard	0.957	0.947	0.947	1.000	0.800

Euclidean Distance

The Euclidean distance is the "ordinary" straight-line distance between two points in Euclidean space. It is sometimes also known as the Pythagorean metric since it uses Pythagoras theorem to calculate the distance and is the most common metric used for high-dimensional distance calculations. The results from using this metric reflect the estimated distances where 2 and 4 are the least similar and 7 and 10 are the most. It even reflects the fact that 2 and 7 are slightly more similar than 2 and 10 which is an important point since distinguishing between these users is vital to creating an accurate visualisation. This was a possible metric.

Hamming Distance

Hamming distance is a metric for comparing two strings or vectors and is simply the number of bit positions in which one differs from the other (Mazda 1993). It is the minimum number of substitutions required to change one string into the other. Using this metric suggested the same pattern of similarity to the estimated distances and Euclidean calculations but represented on a different scale. However it struggled to identify any difference between users 2 and 7 and users 2 and 10 which as mentioned previously is one of the more important identifications required. Furthermore, Hamming distance does not take into account the pairwise difference between points, a vital requirement, only the number of differences across all pairs therefore although the similarities calculated are representative of the data, it was deemed inappropriate for this reason.

Manhattan Distance

Manhattan distance, also known as City Block or Taxicab distance, represents distances between points in a city road grid, or right angles, examining the absolute differences between the Cartesian coordinates of two points (Szabo 2015). To best visualise this, Figure 4.3 shows both the Euclidean distance (green line) and the Manhattan distance (red, blue and yellow line) between two points. All three Manhattan distance lines have the same shortest path length.

Similar results were found to Euclidean and Hamming distance where the pattern of similarities were the same but again on a different scale. Since Manhattan takes into account the Cartesian coordinates of two sets of points and therefore the pairwise differences, this metric was another possible choice.

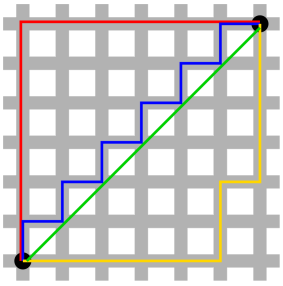


Figure 4.3: *Manhattan distance shown by the yellow, blue and red lines; Euclidean distance shown by the green line to show how their distance is compared (Ranjitkar and Karki 2016).*

Cosine Similarity

Cosine similarity is a measure of similarity where the cosine of the angle between two n-dimensional vectors in an n-dimensional space is computed. Values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar (neo4j 2020). The results from this metric suggest that 2 and 10 are most similar with 2 and 7 being the least. Furthermore it finds 4 and 7 and 7 and 10 to be almost identical in terms of similarity which is not the case. For this reason it was decided that Cosine similarity was not an appropriate metric for this situation.

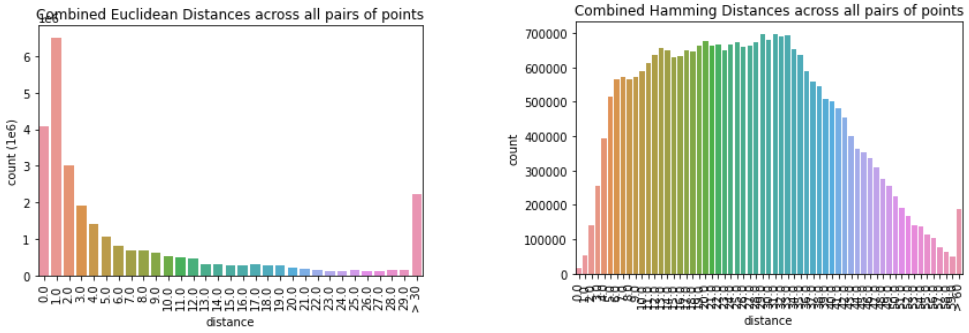
Jaccard Distance

Jaccard distance is a measure of how dissimilar two sets are. It is the complement of the Jaccard Index which compares two sets to see which members are shared and which are distinct. It is also known as the Intersection over Union. The Jaccard distance can be found by subtracting the Jaccard Index from 1 (DeepAI 2019b) where 1 is perfectly dissimilar and 0 is perfectly similar. The results of this metric found that all of these users were relatively dissimilar with users 7 and 10 being the most similar and users 4 and 7 being the least which does not fully correspond to the estimations. This may be down to the fact that only a small number of apps were used by each user which results in many columns equalling 0. Therefore the intersection will be greater for each comparison, since more 0 values occur, drowning out values that are different. This also does not take into account the individual difference between apps instead it takes the dataset as a whole. For those reasons, Jaccard distance was deemed inappropriate for this project.

4.2.2 Evaluation

A key outcome from this experiment was that there is no “perfect” or “best” metric, instead each has pros and cons and is suitable for different situations. To visualise these differences, below are two graphs showing the proportion of distances across all users. Figure 4.4b, Hamming distance, clearly shows a gradual increase, a large, smooth peak and smooth decline. This suggests that there is a large proportion of users far away from each other in terms of app usage. Whereas Figure 4.4a, which used the counts calculated from Euclidean distances shows an initial peak with a sharp decline followed by a gradual plateau suggesting that many users are relatively similar to each other and there are less outliers. Both graphs clearly suggest different ideas as to how these data points are connected to one another.

Since Hamming distance, Cosine similarity and Jaccard similarity were all ruled out, this left just Euclidean and Manhattan distances. Euclidean was favoured over Manhattan since Euclidean distance found the unique shortest path unlike Manhattan where multiple short paths, although not the shortest, were found, illustrated in Figure 4.3. Therefore it was concluded that Euclidean distance was the most appropriate metric for this project.



(a) Euclidean Distance.

(b) Hamming Distance.

Figure 4.4: *Combined distances between every pair of users to show pattern of distances across the dataset.*

Standardised Euclidean

Standardised, or normalised, Euclidean distance is the where the Euclidean distance is calculated between pairwise points and then divided by the variance, given by the following equation (SciPy 2019):

$$\sqrt{\frac{\sum(u_i - v_i)^2}{V[x_i]}}$$

Variance measures how far a set of values are spread out from their mean. In this case, the variance vector, V, is a list of the variance computed over all points for each column, x_i. This solves the problem where Euclidean distance has no maximum bound value leading to problems interpreting the distances computed. Since Euclidean was chosen as the most appropriate metric and Standardised Euclidean is often viewed as more accurate than Euclidean on its own, the decision was made that Standardised Euclidean would be the chosen metric.

4.3 Clustering

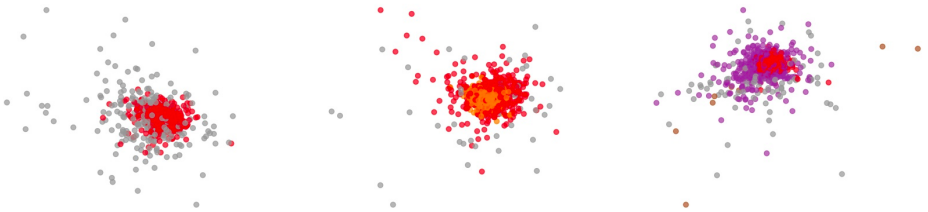
After deciding on the most appropriate metric and before dimensionality reduction algorithms could be used, the next stage was to identify clusters. The reason for this would allow a hypothesis or prediction to be established as to what the visualisations in the next stage would look like. It would also identify if there were clusters or groups within the high-dimensional data in the first place, which were previously not known or discovered before. To do this, a simple k-means clustering algorithm was applied to the high-dimensional dataset. It was found that there were two main cluster groups with a potential other two clusters consisting of outliers. These high-dimensional clusters can serve as features or patterns to watch out for in the low-dimensional layouts, possibly showing strengths and weaknesses of the dimensionality reduction algorithms in question.

4.4 Dimensionality Reduction Algorithms

Once data pre-processing, cluster analysis and metric comparison was completed, the next stage was to combine these for use in conjunction with dimensionality reduction algorithms. This section details the implementation of the algorithms discussed in Section 2.5, specifically the parameters for each algorithm and how the process of parameter selection and optimisation was carried out.

4.4.1 Random Projection

Random Projection was the first algorithm implemented. This algorithm was used primarily as a base layout to compare the more advanced techniques to. However it was also used to visualise the clusters, discussed in Section 4.3, and evaluate whether there were visible clusters of users within the data. Scikit-learn offers multiple in-built methods which implement Random Projection in different ways using different approaches. Sparse Random Projection was favoured for the implementation as it guarantees similar embedding quality to its alternative, Gaussian Random Projection, while being far more memory efficient and allows faster computation of the projected data (scikit learn 2020a). The input was simply the pre-processed data discussed in Section 4.1.



(a) Clusters = 2 *(b) Clusters = 3* *(c) Clusters = 4*

Figure 4.5: *Random Projection layout showing different numbers of high-dimensional clusters.*

The graphs, shown in Figure 4.5, display the outputs from the implementation of this algorithm visualising 2, 3 and 4 clusters with the following parameters selected:

- *n_components* was set to 2 since this was the number of dimensions we were wanting to visualise.
- The other parameters were set to default since this was only a baseline layout and we were not required to find optimal values.

4.4.2 Choosing Parameters

The implementation of Random Projection used all of the default parameter values since this was a baseline layout. However more advanced techniques require careful parameter selection and tuning as small changes can make a significant difference to the overall layout. In order to implement this process fairly and accurately, a second approach was carried out for each of the five other algorithms discussed in Section 2.5; Chalmers' 1996, Hybrid Layout, Pivot Layout, t-SNE and UMAP.

Process

Each of the listed algorithms had to follow the same process for parameter selection. They would first be run with default parameters and then a selection of the main or most important parameters would be chosen and altered individually to find the most appropriate value. These selected values would then be combined and the layout evaluated. These would be judged on visual comparison only meaning the runtime and memory usage would be discarded for this stage as the aim was to find the parameters which would create the most accurate layout. If the layout was satisfactory then the parameter values were accepted else they were altered individually until an accurate layout was created. Each algorithm will now be discussed, including the possible parameters and the result of each experiment.

Distance Metric

This parameter is the only one that remains constant throughout this experiment and is used in every algorithm. This is the metric used to calculate the distance between points during runtime of the algorithm. As discussed in Section 4.2, there are many metrics to choose from which is why it was vitally important that prior analysis was carried out to find the most appropriate metric for this dataset. As a reminder, Standardised Euclidean was chosen to be this.

4.4.3 Spring Models

For this section, three different algorithms were implemented:

- Chalmers' 1996 Algorithm
- Hybrid Layout Algorithm
- Pivot Layout Algorithm

As mentioned, a previous implementation of these three algorithms was used as a starting point (Cattermole 2019). This existing implementation, in particular parameter selection, was created with the idea for use with alternative datasets. Although since every dataset is different, some alterations were still required in order for it to be suitable for use with this app usage dataset. For each algorithm there are many parameters and as per the experiment process, only the most important parameters were chosen to assist in creating the most appropriate layout.

Iterations

Perhaps the most important parameter, iterations, in this case, is the number of times the algorithm is run and the forces between points are calculated. Too little iterations results in an output which is incomplete whereas too many iterations creates an inefficient algorithm. It is important to note that there is no optimal number of iterations, regardless of dataset, instead we can only attempt to optimise or improve the efficiency of the algorithm by choosing a value which strikes a balance between the two extremes mentioned. One method of finding this balance is to plot the average velocity of all the points in the graph layout against iterations, as shown in Figure 4.6.

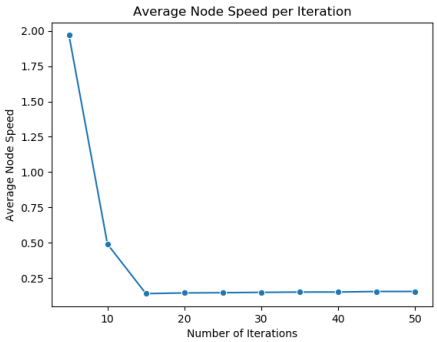


Figure 4.6: *The average velocity of each point in the graph against iterations.*

It can be seen that velocity starts high with an initial sharp decline then begins to plateau out as the number of iterations increases. This is a typical pattern with the rare exception of some algorithms where some phases may be faster, or slower, than others. It is in this plateau section where we can say that the graph has settled as it is typically bound within a small range. This small range represents virtually no change in the output itself; instead it is simply tiny variations. Ideally we would eventually like to see velocity be reduced to zero however the design of Chalmers' 1996 and its successors prevent this. Since these algorithms use stochastic random sampling, tiny

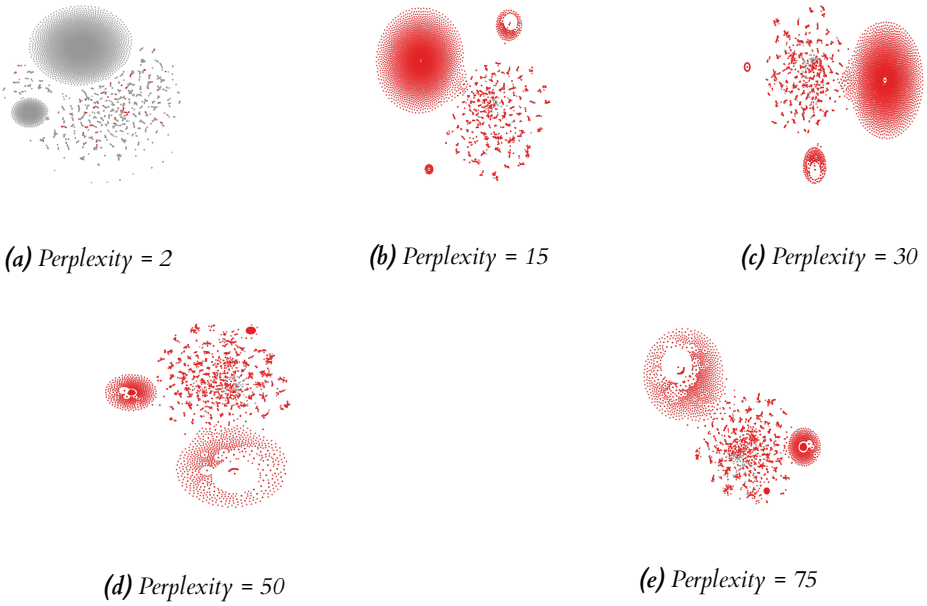
forces are applied on every iteration in order to break out of a potential local minima therefore velocity will never reach zero. Surprisingly we can see this plateau section occurs around 10–15 iterations with the average node speed settling at around 0.15. The existing implementation of these algorithms allowed a termination condition to be set where the algorithm would terminate once average node speed reaches a specified value. Based on these results, this value was set to 0.15. This removes the need to choose a fixed number of iterations although the maximum number of iterations would be set to 50 in such a case where this may be required.

4.4.4 t-SNE

Similar to Spring Models, the number of potential parameters is great, and the output can be altered dramatically through a few small changes. For this reason, the second experiment was carried out with t-SNE where only the most important parameters were altered to find the most appropriate output.

N_components
This parameter was set to the default value of 2 since we want to reduce our dataset to 2 dimensions.

Perplexity
Perplexity deals with balancing the attention between global and local aspects of the data. It is essentially a guess about the number of close neighbours each point has (Wattenberg et al. 2016). Figure 4.7 shows a range of layouts each created with a different value of *perplexity*.



(a) *Perplexity = 2* ***(b)*** *Perplexity = 15* ***(c)*** *Perplexity = 30*

(d) *Perplexity = 50* ***(e)*** *Perplexity = 75*

Figure 4.7: *t-SNE layouts with different values of perplexity.*

From this figure, we can see that a value of 2 and 75 create 2 clear clusters however the central area is rather confused and cluttered and there are no real identifiable clusters. A value of 50 slightly improves on this however it is poor when compared to values of 15 and 30. These two

options create relatively similar layouts but ultimately the value of 30 was chosen as the most appropriate due to the slightly tighter central area.

N_iter

Unlike with Spring Models, it is not possible to plot a graph of velocity against iterations to find an appropriate number of iterations. Instead, the number of iterations was set to 5000, for such a case where this may be necessary. However, this is combined with another parameter, *n_iter_without_progress*, which was set to the default value of 300. This value is the maximum number of iterations without progress before optimization is aborted meaning if the output has not changed after 300 iterations, then the algorithm will finish. This is beneficial as it means the algorithm is not required to run all 5000 iterations, saving time and memory usage when possible.

Early_exaggeration

Early_exaggeration controls how tight natural clusters in the original space are in the embedded space and how much space will be between them (scikit learn 2020c). The value chosen for this was 1. Figure 4.8 shows the layout with this chosen value comparing it to the default value of 12. It can be seen that the chosen value of 1 creates a layout which is slightly more spread out in terms of large clusters than the other, which represents the default value of 12. Since the aim is to create an informative visualisation and clusters can be viewed clearer when they are further apart, this choice of value is justified.

(a) *Early_exaggeration = 1* ***(b)*** *Early_exaggeration = 12*

Figure 4.8: *t-SNE layouts with different values of early_exaggeration.*

4.4.5 UMAP

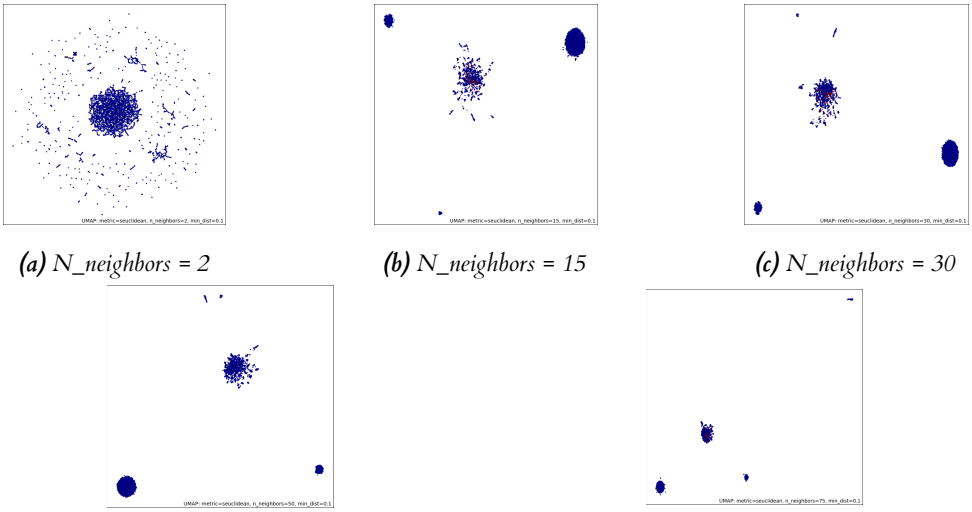
There are four main parameters for UMAP; *n_components*, *n_neighbors*, *min_dist* and *metric*. Although there are several others, 29 to be precise (UMAP-Learn 2020), as explained in the experiment process and to keep decisions consistent throughout the project, these were the four parameters that would be concentrated on and tuned.

N_components

Similar to previous algorithms, this parameter was set to the default value of 2 since we want to reduce our dataset to 2 dimensions.

N_neighbors

Similar to *perplexity* with t-SNE, this parameter balances local versus global structure within the data. Figure 4.9 shows the difference between layouts when this parameter is changed.



(a) N_neighbors = 2 *(b) N_neighbors = 15* *(c) N_neighbors = 30*

(d) N_neighbors = 50 *(e) N_neighbors = 75*

Figure 4.9: *UMAP layouts with different values of N_neighbors.*

It can be seen that a range of layouts are produced as the size of *n_neighbors* increases. A value of 2 shows only the local structure where only small clusters are shown and points are typically evenly spaced apart. Conversely a value 75 shows only the global structure where only large clusters are shown, rendering both inappropriate. Values of 15 and 30 both show a good balance however the central cluster is quite expanded even though it shows local clusters well. As a result it is unknown whether some small clusters are part of the larger cluster or if they are separate. A value of 50 finds a balance here showing the global structure well, separating into distinct clusters as well as displaying small clusters within these large clusters, particularly in the centre cluster. Therefore, 50 was chosen as the most appropriate value.

Min_dist

This variable controls how tightly UMAP is allowed to pack points together. As the name suggests, it dictates the minimum distance apart that points are allowed to be in the low-dimensional representation. This parameter is set relative to the parameter spread which defines how clustered or clumped the embedded points are (UMAP-Learn 2020). These were set to 0.75 and 3 respectively. Figure 4.10a shows the default values of 0.1 and 1 and Figure 4.10b shows the chosen values. It was decided that larger values of *min_dist* and *spread* were required as this would result in a more even dispersal of points thus allowing local clusters to be identified more easily.

(a) Min_dist = 0.1, spread = 1 *(b) Min_dist = 0.75, spread = 3*

Figure 4.10: *UMAP layouts with different values of min_dist.*

4.4.6 Evaluation

This experiment has outlined the vast amount of potential parameters across Spring Models, t-SNE and UMAP. For each algorithm, the most important parameters were selected and tweaked in order to find the value which would contribute to the most appropriate visualisation. Standardised Euclidean was the distance metric for each algorithm and this remained constant throughout each experiment. For Spring Models, the parameter *iterations* was given the value of 50 and for each spring model, algorithm specific parameters were given their default value, as recommended and used in Chalmers; Morrison et al. (2002); Morrison and Chalmers (2003). t-SNE had 4 main parameters; *n_components*, perplexity, *n_iter* and *early_exaggeration* which were given a value of 2, 30, 5000 and 1 respectively. Similarly, UMAP had 4 important parameters; *n_components*, *n_neighbors*, *min_dist* and *spread* where the most appropriate values were found to be 2, 50, 0.75 and 3. An important point to note is that these chosen values may not be optimal however, similar to Section 4.2, it may be argued that there are in fact no optimal values making this a challenging problem. Every algorithm and set of parameters produce different layouts therefore the best we can do is to subjectively optimise this layout as best we can. This is what this experiment sought out to achieve.

4.5 Command Line

An existing feature in the base template mentioned in Section 2.3 was that all algorithms could be called via the command line. This was beneficial as this feature is fundamental to the process of assessment and analysis of the core layout algorithms. It was decided to keep and improve this feature to increase usability and the depth of analysis which could be achieved. Each algorithm has its own command line signature which is explained below however they all follow the following format:

```
$ python3 algorithm.py [num apps] [dataset size] *OPTIONAL* [extra parameters]
```

This was feature 1. *num apps* is the number of apps and therefore dimensions that will be taken into consideration when the distance between users is calculated. This value is one of [100, 500, 1000, 10000, 45788]. *dataset size* is the number of users that will be compared in the range of [100, 500, 1000, 7504, 10368]. Since the extra parameters for each algorithm changes, it was important to provide the ability for the user to view and experiment with these valid parameters. This was feature 2. This is done by simply calling the algorithm with no added parameters:

```
$ python3 algorithm.py
```

Appropriate error handling was another important requirement to have in the absence of a GUI where this is usually provided. If invalid parameters are provided then the appropriate message is displayed with a suggestion on how to fix the error. This prevents invalid parameters being passed to the algorithm resulting in a crash of the program making it feature 3. The final feature implemented was to notify the user what algorithm is being run along with its parameters, as well as displaying metrics once the algorithm was complete. These metrics include:

- Runtime
- Memory usage
- Layout time

Examples of all four of these features can be seen in Figure 4.11.

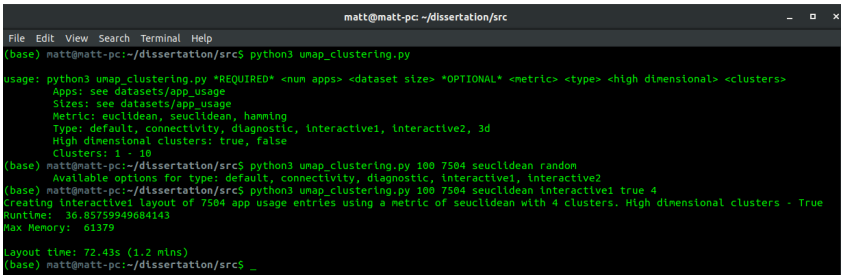


Figure 4.11: *Command Line call demonstrating error handling and information display.*

The first command line call shows feature 2 where the parameters for the chosen algorithm are displayed. The second call shows feature 3 where "random" is not an appropriate value for the *<type>* parameter and so the valid parameters are listed. Feature 1 is shown in call 3 where the algorithm is run using all of the valid parameters and feature 4 is shown in the output where information regarding the layout is displayed.

4.5.1 Spring Models

There are 5 extra parameters for Spring Models. The aim of these is to improve the range of visualisations available to the user. The 5 parameters explained below can be seen in Figure 4.12.

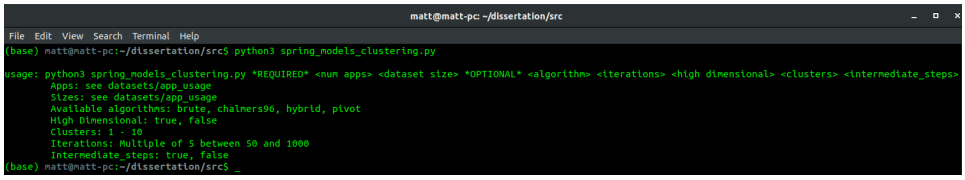


Figure 4.12: *Command line call for Spring Model algorithms.*

algorithm
This command line argument specifies what Spring Model algorithm will be used out of the three discussed; Chalmers' 1996, Hybrid Layout and Pivot Layout.

iterations
This command line argument specifies the maximum number of iterations on the occasion that the user would like to experiment with different values other than the default of 50, up to a maximum of 1000. However values greater than 50 will only be valid in situations where average node speed does not reach its limit of 0.15, discussed in Section 4.4.3, within 50 iterations .

high dimensional
This command line argument allows the user to choose between what clusters are shown in the visualisation. High-dimensional clusters are the clusters found using k-means clustering on the original high-dimensional dataset, the true clusters in the data, as mentioned in Section 4.3. Whereas low-dimensional clusters are the clusters found when k-means clustering is applied on the low-dimensional data, or the output of the dimensionality reduction algorithm. An example of this feature can be seen in Appendix A.2 highlighting the difference between low and high-dimensional clusters.

clusters
This command line argument allows the user to specify the number of clusters they would like to visualise in the graph. This value can be set from 1 to 10 however as mentioned in Section 4.3, 2-4 clusters seemed to be the most appropriate value dependent on the size of the dataset used. This feature simply allows the user to experiment and the output can be shown in Figure 4.5 using the Random Projection algorithm to visualise different clusters.

intermediate steps
This command line argument allows the user to display the layout of the algorithm at different stages during runtime, specifically in increments of *iterations/5*. The purpose is to allow further investigation into how the algorithm works by breaking it into 5 layouts and having the ability to analyse how the graph changes after every *iterations/5* iterations. This feature can be seen in Appendix A.3. It is important to note that unlike the other features, this one was not used during the algorithmic performance evaluation as it is simply an additional feature to aid visual analysis of the algorithms.

4.5.2 t-SNE

Figure 4.13 shows the command line call for t-SNE. There are five extra parameters where four are similar to Spring Models; *iterations*, *high_dimensional*, *clusters* and *intermediate_steps*. The fifth argument was *metric*. This was replaced for the parameter *algorithm*, to give the opportunity to experiment with different distance metrics. This option was not available for Spring Models as it was decided it was important to limit the number of extra parameters without impacting analysis, and it was found that *algorithm* was more beneficial to Spring Models than *metric* since there were multiple Spring Model algorithms to choose from.

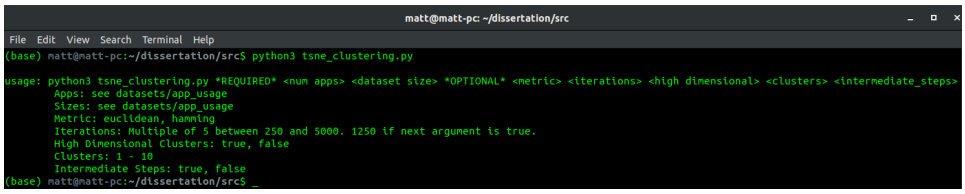


Figure 4.13: *Command line call for t-SNE.*

4.5.3 UMAP

Figure 4.14 shows the command line input for this algorithm. It is similar to the t-SNE call with the exception that *iterations* and *intermediate_steps* have been omitted as these are not valid for UMAP. Instead a new argument, *type*, is available.

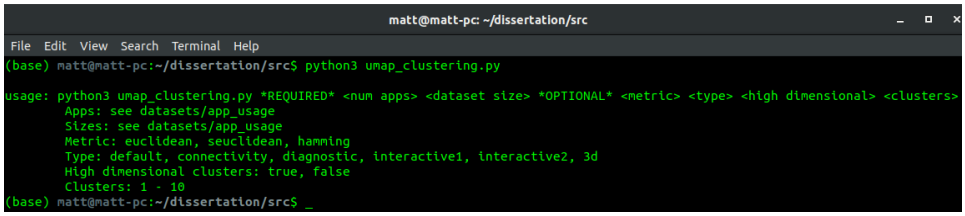


Figure 4.14: *Command line call for UMAP.*

type
This command line argument defines what type of graph is output. The options are a default layout of UMAP, a default layout showing how well local neighbourhoods are preserved through the means of the Jaccard Index (diagnostic), a layout connecting all data points together (connectivity), two different versions of an interactive layout where points are annotated and lastly a 3D layout. Each graph exists for a different purpose of analysing and investigating. All six of these layouts can be viewed in Appendix A.4.

4.6 Summary

This section addressed the process of how the project's design requirements were met from firstly pre-processing the data to finding the most appropriate metric then clustering and implementing the algorithms and lastly visualising the output. Two approaches were taken to streamline the process; one for choosing the most appropriate metric, which was found to be Standardised Euclidean, and one for finding the most appropriate parameters for each algorithm since they have the ability to dramatically alter the layout produced. Both approaches were planned and carried out carefully to ensure fair comparisons for the evaluation. Each algorithm could be called from the command line and the signature of all three were outlined as well as extra parameters like *high_dimensional*, *clusters* and *intermediate_steps*. The purpose of these extra parameters was to increase the information that could be gathered from the data through alternate visualisations.

5 Evaluation

The main goal of this evaluation is to compare the visualisations created by Spring Models, t-SNE and UMAP in terms of runtime, memory usage and visual accuracy. Random Projection will also be included as a baseline. This chapter will discuss the setup of the evaluation, the methods used to ensure fair experiments, and the results of the evaluations.

5.1 Setup

This section will discuss the process of planning the evaluation, in particular the decisions made with regards to the hardware used and its specifications.

5.1.1 Hardware

Throughout testing of the algorithms, it was clear that runtimes would greatly increase as the size of the dataset increased. As this exact increase was unknown prior to the evaluation, the use of a personal laptop was ruled out for running the evaluation as a result of this uncertainty. It was thought that a personal laptop would not be able to handle multiple resource hungry layouts for any length of time. In order to ensure that every run of each algorithm was given the same resources to produce a layout, a high-end PC was used, with the algorithm being the only process running. This custom built PC was equipped with an AMD Ryzen 5 processor running at 3.6GHz and 16GB RAM. To ensure consistency, all comparisons were run sequentially to ensure full allocation of resources to the algorithm should it require it. This would also remove any bias to the evaluation. After each algorithm was run, results were recorded for analysis.

5.2 Parameter Selection

As mentioned in each of their respective sections, each algorithm has many parameters. In order to perform a fair comparison between them, it was important to consider each parameter carefully before selecting the most important. Every parameter would be classed as either an independent, dependent or control variable. This section will discuss the classification of variables.

5.2.1 Control Variables

The control variables were kept constant for all visualisations to ensure a fair evaluation.

Dataset
All layouts were created using the same dataset, App Usage, introduced in Section 2.6. This was important so that the results could be directly compared. The distance metric used in each algorithm was kept constant also as this can greatly affect the visual output, runtimes and memory usage. The dataset was split into different subsets however only direct comparisons were made with same size datasets to ensure fairness and accuracy throughout the evaluation.

Table 5.1: *The values assigned to each specific parameter for each algorithm*

Algorithm	Parameter	Value
Spring Models	iterations	50
	metric	Standardised Euclidean
t-SNE	n_components	2
	perplexity	30
	early_exaggeration	1
	n_iter	1000
	metric	Standardised Euclidean
UMAP	n_components	2
	n_neighbors	50
	min_dist	0.75
	metric	Standardised Euclidean

Algorithm Specific Parameters
Each algorithm has many specific parameters that had to be chosen and tuned. The goal when selecting these parameters was to select the ones that would allow the algorithm to perform to its best possible standard on our dataset. This was important as there are no constant optimal parameters, instead, they vary based on the metric and dataset used. A recap of these algorithm specific parameters and their values are shown above, in Figure 5.1.

5.2.2 Independent Variables

The independent variables in this case were the algorithm being used and the size of the subset being visualised. Each algorithm: Chalmers96, Hybrid Layout, Pivot Layout, t-SNE, UMAP and the baseline Random Projection, described in section Section 2.5 were run with varying sizes, N, between 100 and all users; N = [100, 500, 1000, 7504, 10368]. D, the number of apps, was also taken into account with five different values ranging from 100 to all apps; D = [100, 500, 1000, 10000, 45788]. Each comparison, the algorithm run on a subset, was repeated three times to account for variance within the results.

5.2.3 Dependent Variables

The dependent variables for this evaluation were the performance metrics discussed in Section 2.7; runtime, memory usage and the overall visual structure of the layout. The first two were measured and the third evaluated, for each layout so that a direct comparison could be made between algorithms and subsets. It was important that each of these were measured correctly and precisely.

5.3 Results

Once the visualisations had been created and the metrics obtained, the comparison and analysis of results could begin. This section summarises these results, in the form of graphs and visual layouts, and will discuss the findings.

5.3.1 Visual Results

Although individual visual evaluations were carried out in Section 4.4.2 while selecting the most appropriate parameters for each algorithm, this section looks at comparing the layouts collectively. The figures below show the layouts produced by each algorithm with the following conditions:

- N = 7504; active users as mentioned in Section 4.1.2;
- D = 100; most popular apps
- 2 high-dimensional clusters
- Tuned parameters, see Table 5.1

Using these conditions allows for a direct and fair comparison to be made between visualisations. As reminder, each layout will be examined based on the following criteria:

- The accuracy of high-dimensional clusters on the low-dimensional layout
- The overall position and accuracy of data points, including distance between points and clusters

Random Projection

Random Projection was used as a visual baseline for this evaluation which allowed more advanced techniques to be compared to it. It was also used to visualise high-dimensional clusters in a low-dimensional space to confirm the existence of them.

This baseline layout, shown in Figure 5.1a, proves the existence of clusters whereby there is a central collection of points, in red, surrounded by another cluster, in grey, although the whole layout is rather clumped together. Analysis of the graph, through the softwares ability to zoom into clusters, confirmed there to be only some overlap as the centre cluster consists almost fully of red points with the surrounding areas only grey points. However this is to be expected since there is a trade off between accuracy for efficiency. This layout preserves high-dimensional distances as documented however it does not make use of distance metric to calculate more accurate distances between points. However it succeeds in its main purpose of visualising clusters since they are based on high-dimensional distance and it can be seen that these are preserved relatively accurately in the low-dimensional space.

(a) *Random Projection algorithm.* ***(b)*** *Chalmers' 1996 algorithm.*

Figure 5.1: *The layout produced from Random Projection and Chalmers' 1996 algorithms of 7504 data points, showing 2 high-dimensional clusters.*

Chalmers' 1996

Chalmers' 1996 was the first of the three Spring Model algorithms implemented. This algorithm creates a similar layout, shown in Figure 5.1b, to that of Random Projection with some distinct differences. The layout is far more spread out than Random Projection and this may be attributed to the use of a distance metric by Chalmers' 1996 algorithm to more accurately calculate distances. The layout visualises similar clusters to Random Projection also with a dense central cluster and what can only be explained as a cluster ring around the outer area.

Hybrid Layout

Hybrid Layout algorithm was the next Spring Model implemented. This was a computational improvement to Chalmers' 1996 algorithm and so it was hoped that this would maintain or improve the accuracy of the visualisation produced by Chalmers' 1996.

This layout can be seen in Figure 5.2a and again improves on the previous two layouts from Chalmers' 1996 and Random Projection. Instead of just a centre circle of points, which suggests a potential generalisation, this layout shows a more refined shape where points seem to be located in more specific positions than that of the previous layout. This may be attributed to the use of interpolation in the Hybrid Layout algorithm where neighbours are found more accurately. The specific positions of points seems to be the only alteration since the same pattern of clusters appears with a dense central cluster, although more refined, surrounded by a second cluster with some overlap between points.

(a) *Hybrid Layout algorithm.* ***(b)*** *Pivot Layout algorithm.*

Figure 5.2: *The layout produced from Hybrid Layout and Pivot Layout algorithms of 7504 data points, showing 2 high-dimensional clusters.*

Pivot Layout

Pivot Layout was the third and final Spring Model implemented. With this being a successor to Hybrid Layout it was hoped this would result in an improved layout.

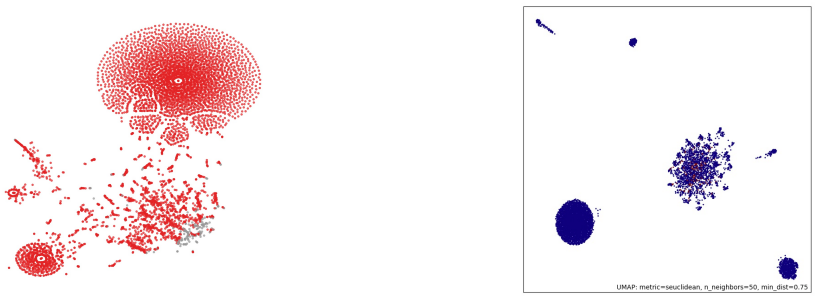
As seen in Figure 5.2b, there is a significant difference between the layout produced by this algorithm and its predecessors. There is a distinct linear feel to this layout almost as if points are located along an imaginary central axis. This may be down to the logic of the algorithm where pivots are chosen and nodes are positioned around these pivots by best estimating what surrounding bucket they belong in. A reminder that there are $N^{1/4}$ buckets. This essentially groups points into sectors which is potentially what has occurred here. The only similarity seems to be the pattern of high-dimensional clusters with a dense centre cluster and a surrounding second cluster.

t-SNE

The next algorithm was t-SNE. Since Chalmers' 1996, Hybrid Layout and Pivot Layout all belong to the same class of dimensionality reduction algorithms; Spring Models, it was expected that their produced layouts may be of similar shape and pattern. However as t-SNE uses a completely different method, Section 2.5.3 , it was presumed that a different visualisation would be created and is shown in Figure 5.3a.

This is a significantly different layout to that of Spring Models. There seems to be 4 or 5 main

clusters of points within the layout however not all are distinct since there is some overlap. A known drawback of t-SNE is that it struggles to preserve global structures in the data where distances between clusters are rarely accurate nor do they mean anything (Wattenberg et al. 2016). This is a possibility here where local structures within clusters seem to be well preserved but the distances between these larger clusters do not. However high-dimensional clusters have been visualised and plotted well since the previously mentioned outer circle consisting of points belonging to the same cluster, visualised in previous layouts, are positioned together in the bottom right of the layout, part of a larger cluster.



(a) *t-SNE algorithm.* ***(b)*** *UMAP algorithm.*

Figure 5.3: *The layout produced from t-SNE and UMAP algorithms of 7504 data points, showing 2 high-dimensional clusters.*

UMAP

After evaluating the layout of t-SNE and identifying potential drawbacks, it was interesting to compare it to UMAP to see whether UMAP was able to improve on these drawbacks. The layout produced is shown in Figure 5.3b.

The first major noticeable difference in the preciseness of clusters. There seems to be 5 or 6 clear clusters in this layout. However the cluster shapes are not dissimilar to those of t-SNE with a large central cluster of separated but connected points, two dense smaller clusters and two additional smaller clusters, in the shape of a circle and straight line. The key difference here is that UMAP seemingly preserves these distances between clusters and overall global structure much better than t-SNE showing distinct large clusters of different distances to one another. Local structure seems to be preserved well also with smaller clusters of points within these larger clusters, in particular the central cluster. The high-dimensional clusters themselves are not as clearly visualised although the majority of the original outer clusters' points are positioned in the centre of the central cluster, outlined in red.

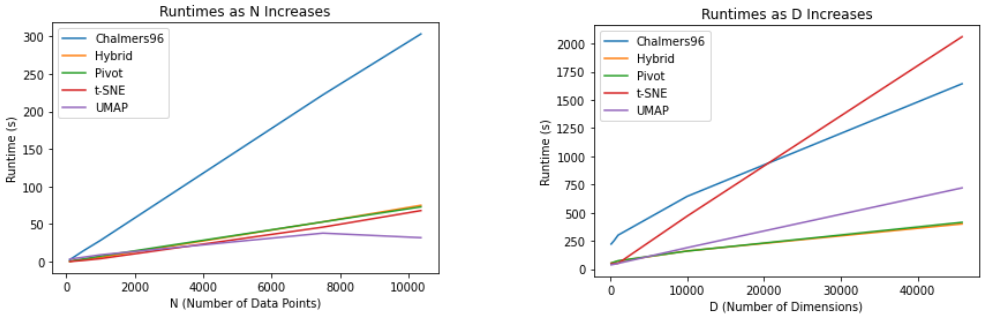
Conclusion

All six layouts, shown using the same conditions, produce slightly different visualisations, each with pros and cons. Random Projection visualises high-dimensional clusters well but struggles to separate points, resulting in a clumped layout. The three Spring Models; Chalmers' 1996, Hybrid Layout and Pivot Layout again show high-dimensional clusters in the same pattern with a central clumped cluster surrounded by an outer cluster. However each algorithm plots points in different locations in the 2D space, resulting in different overall cluster shapes. t-SNE identifies clusters and preserves local structure within the data well however struggles to preserve the global structure and it is known that these clusters may not always be true. UMAP seemingly preserves global and local structures well identifying distinct clusters but struggles to show high-dimensional

clusters as well as some of the other algorithms. Overall, each of these algorithms can be seen as being limited in their own specific ways. The next stage will look at the cost of creating these visualisations, in terms of runtime and memory usage, and whether trade offs are made between them in order to create these layouts.

5.3.2 Runtime Results

In order to fully evaluate all algorithms in terms of runtime, two different approaches were taken. The first approach looked at comparing runtimes as the number of data points increased and the second looked at comparing runtimes as the number of dimensions increased. As dimensions and data points are independent variables, it was decided that the most accurate and fairest way was to compare them separately.



(a) *N is altered, D = 100*

(b) *D is altered, N = 7504*

Figure 5.4: *Effect on runtime as N and D are altered independently with the omission of Random Projection*

Increase N

Using the parameters listed above for each algorithm, Figure 5.4a illustrates the runtimes of each algorithm as N, the number of data points, is increased and D, number of dimensions, remains constant at 100, which take into account the 100 most popular apps.

It was decided that Random Projection would not be shown on the graph due to its efficiency. It was found that the runtime for this algorithm was a maximum of 0.004s and therefore it would not be useful to visualise on the graph. This was expected to be the quickest algorithm since it is designed for efficient computation although this is caused by a tradeoff for accuracy which can be seen in its visual layout above.

It can be seen that Chalmers' 1996 algorithm increases linearly as the number of data points are increased, running at a time of 303s for N = 10,368. It was expected that this would be the slowest algorithm since it is the oldest and successors like Hybrid and Pivot layout were created to improve upon it.

Hybrid and Pivot Layout were similar in terms of runtime. Hybrid outperformed Pivot slightly for the first three comparisons; N = 100, 500 and 1000, then posted identical times for N = 7504 of 53s. Pivot Layout then outperformed Hybrid Layout in the last comparison, where N = 10,368, by 2s. These are very minimal differences which confirms these two algorithms are similar in terms of complexity, as mentioned in Section 2.5.2, they are $O(N\sqrt{N})$ and $O(N^{5/4})$ respectively. Pivot Layout performed better when more data points were available which may suggest that if the dataset consisted of more than 10,368 data points then maybe the Pivot Layout algorithm would increase the difference in runtime between the Hybrid Layout algorithm. A further fair

experiment would need to be carried out with a greater number of data points to confirm this theory.

t-SNE improved upon the runtimes of Hybrid Layout and Pivot Layout algorithms. Using the dataset where N = 7504, this was run in a time of 46s, improving upon the best Spring Model runtime by 7s. Again this was expected since the runtime complexity of t-SNE ($O(N \log N)$ is better than that of Pivot Layout ($O(N^{5/4})$).

UMAP then improved on the runtimes of t-SNE when more data points were compared. Initially runtimes were the slowest of all algorithms registering 3.5s for N = 100 compared to the next slowest of 2.7s using Chalmers' 1996. As data points increased, the time difference decreased although still slower than that of Hybrid Layout, Pivot Layout and t-SNE algorithms, until 7504 data points where UMAP finished in 38s compared to Spring Model's best of 53s and t-SNEs best of 46s. Surprisingly when 10,368 data points were used with UMAP, a faster time of 32s was recorded to that of 7504 data points where 38s was recorded. A potential reason for this is that these extra users require less time to calculate their nearest neighbours since there is a good probability they are all similar, as they were all removed due to being classed as inactive. These results can be viewed in table format in Appendix A.1.

Increase D

Figure 5.4b and Appendix A.2 shows the results of the second runtime comparison where the D is increased and N, remains constant at 7504. It can be seen that, similar to the first comparison and as expected, Chalmers' 1996 algorithm is slowest taking 1640s, over 27 minutes, to finish running. Again, both Hybrid and Pivot Layout algorithms took around the same time to finish running, with Hybrid Layout slightly outperforming Pivot Layout by around 14s when D = 45788 (401s v 415s). Since N = 7504 in this case, similar to the reasoning above, the theory is that Pivot Layout begins to outperform Hybrid Layout once N > 10368 which could explain why Hybrid Layout performed the best here. Surprisingly, Pivot and Hybrid Layout both completed running in a faster time than t-SNE and UMAP. Both t-SNE and UMAP performed better when the value of D was lower. They performed better when D = 100, 500 and 1000, recording times of 38s, 43s, 51s (UMAP) and 46s, 48s, 51s (t-SNE) compared to the fastest Spring Model times of 53s, 60s and 70s. However the big increase to D = 10,000 likely decreased performance resulting in times of 190s for UMAP and 470s for t-SNE compared to the 160s runtime recorded by the Pivot Layout algorithm. It was expected that UMAP and t-SNE would have been able to handle this large increase in D and still outperform Spring Models. This is further discussed in Section 5.4.

Conclusion

These results show the large differences between runtimes across all algorithms although these were expected. Random Projection was the most efficient however due its major tradeoff between runtime and visual accuracy, it was omitted to ensure equal comparisons. Chalmers' 1996 algorithm was the least efficient with Hybrid Layout and Pivot Layout improving upon it with almost identical results. However it must be noted that Pivot Layout is designed to be more efficient than Hybrid Layout when dealing with a large number of data points and there is a chance that N = 10,368 was too small considering they have previously run successfully on over 1 million data points (Cattermole 2019). t-SNE and UMAP improved on these Spring Model runtimes when the value of N was increased which was expected due to the improved complexity of these algorithms. However both algorithms slightly lacked efficiency when D was increased which was unexpected resulting in an additional approach documented in Section 5.4.

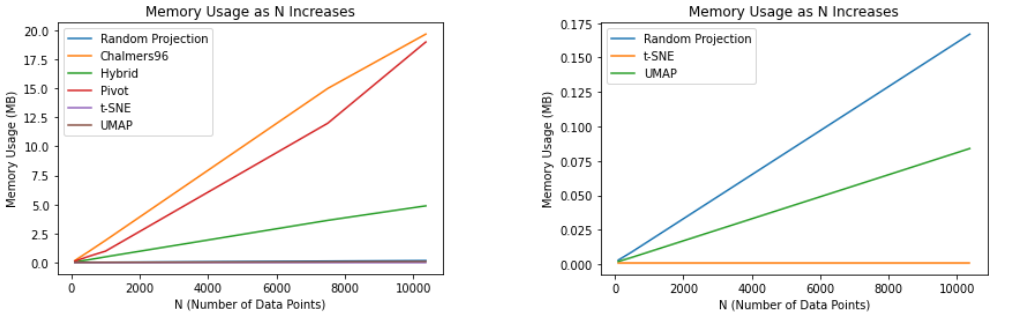
5.3.3 Memory Usage Results

Similar to the runtime evaluation, two different approaches were taken; one where only the value of D was altered and the other where only the value of N was altered. Again, the aim was to

create a fair comparison where only independent variables were changed and their effect evaluated.

Increase N

Again using the chosen parameters listed in Table 5.1, N, the number of data points, was changed while the number of dimensions, D, remained constant at 100. Figure 5.5 shows the results in graphical form with Appendix A.3 showing a table form of the same results.



***(a)** All algorithms.* ***(b)** Only Random Projection, t-SNE and UMAP.*

***Figure 5.5:** Effect on memory usage as the size of the dataset increases.*

Figure 5.5a shows the results of all six algorithms however due to the large memory usage in Spring Models, the graph cannot show the trend of the three other dimensionality reduction algorithms. Therefore, Figure 5.5b shows the memory usage of just Random Projection, t-SNE and UMAP.

As expected Chalmers' 1996 is the most inefficient algorithm showing a linear increase as N increases. For N = 10,368, Chalmers' 1996 used on average 19.7MB. This large memory usage can be attributed to the caching involved where up to five nodes and their distances are stored in the neighbour set for each node as described in 2.5.2. Since Hybrid Layout was a successor of Chalmers' 1996 and it involved no caching, it was hoped this would decrease the memory usage required to run the algorithm. This was the case and Hybrid Layout used a maximum of 4.9MB when N = 10,368. For the same value of N, Pivot Layout used 19.0MB which is similar to Chalmers' 1996 and a large increase compared to Hybrid Layout. It is important to note that 19.0MB was the average memory usage for Pivot Layout as memory usage changed dramatically between runs. A potential reason as to what caused this is that Pivot Layout pre-calculates and caches distances between nodes and the pivot set, as discussed in Section 2.5.2, in order to decrease complexity during the interpolation stage. It was thought that this random pivot set was the cause of the this variation in memory usage however a further experiment would have to be carried out to prove this. Moving on to the other three dimensionality reduction algorithms, Random Projection is an efficient algorithm with regards to runtime but also in terms of memory usage using a maximum of 0.167MB when N = 10,368, as shown in 5.5b. A large decrease to even the most efficient Spring Model. It was expected that this algorithm would be the best performer however it was proved that t-SNE and UMAP both required less memory. UMAP used a maximum of 0.084MB to run the algorithm whereas t-SNE required a constant 0.001MB for every value of N tested.

Increase D

Appendix A.4 and Appendix A.13 shows the results of the second approach where only D was altered to one of the values of [100, 500, 1000, 10000, 45788] while N remained constant

at 7504. Again for the purpose to improve analysis, Figure A.13a shows all 6 algorithms and Figure A.13b shows only Random Projection, t-SNE and UMAP. It can be seen that for every algorithm, with the exception of Pivot Layout, the memory usage required to run remained constant as the value of D increased. Chalmers' 1996 required 15MB, Hybrid Layout 3.6MB, t-SNE 0.001MB and UMAP 0.061MB. The memory used by Pivot Layout changed for each run and this was attributed to the same potential reason as before where random pre-calculated distances are used to make the algorithm more computationally efficient. It was therefore difficult to obtain a fixed number and so an average was taken over three runs. These constant values are as a result of each algorithm only requiring memory as the number of users, N, increases since the users and their location in the low-dimensional space are the only aspect needing to be stored. The number of apps, D, is only taken into account during the distance calculation between points.

Conclusion

In conclusion, the results here show that the memory usage of each algorithm varies dramatically although they can all be explained by their logic. The use of caching within Chalmers' 1996 algorithm and potentially the pre-calculation of distances within the Pivot Layout algorithm result in greater memory usage compared to Hybrid Layout where there is no use of caching. If this caching was removed it would be expected that memory usage would be similar to that of Hybrid Layout however this would increase runtimes and so this is an acceptable tradeoff. However there is a significant difference between memory usage of Hybrid Layout and both t-SNE and UMAP which makes them more feasible for usage with larger datasets.

5.4 t-SNE v UMAP

This evaluation created some interesting discussion points about the two more advanced dimensionality reduction algorithms, t-SNE and UMAP. It was expected that both algorithms would outperform Spring Models in all comparisons however this was not the case. Both algorithms, although especially t-SNE, struggled to produce efficient layouts where dimensions were extremely high (>10,000) and a theory arose as to why this was the case. Like any other dimensionality reduction algorithm, parameters play a crucial role in the visual layout. Some of these parameters may sacrifice runtime to achieve better visual accuracy and this was a suggested occurrence here. To confirm or deny this theory, a different approach was taken.

5.4.1 Default v Tuned

This approach involved only t-SNE and UMAP where their runtimes were recorded using all default parameter values and the tuned parameter values outlined in Table 5.1. A baseline including the recorded best runtime, from Hybrid Layout, was included also to show, if any, improvement. Figure 5.6 shows this comparison.

For UMAP, using default parameters significantly decreased runtime from 720s when D = 45788 to 190s. It was found that this was caused by one parameter in particular, *n_neighbors*. The default value for this parameter is 15 whereas it was found in Section 4.2 that to achieve the most appropriate output for this data, this value should be set to 50, clearly making a huge difference in efficiency. If this was compared to the runtime of the fastest Spring Model when D = 45788, UMAP outperformed it by around 52% (190s v 401s) showing the efficiency of UMAP when the goal is to minimise runtime.

t-SNE actually performs worse when default parameters are used. *Perplexity*, similar to *n_neighbors* for UMAP, is a common parameter that increases runtimes when it is increased since more neighbors have to be found for each point however the default value of 30 is also the value found to be the most appropriate therefore this is not the reason for this increase in runtime. This leaves the other tuned parameter, *early_exaggeration*, as the reason for this slight increase. The default

value is 12 whereas the most appropriate value was found to be 1 which also seems to have a positive impact on runtime. Therefore unlike UMAP, using default values does not improve efficiency and so parameter selection was not the reason that Spring Models outperformed t-SNE for high values of D. This leads to the conclusion that potentially t-SNE does not scale well, in terms of runtime, with extremely high dimensions. This was backed by van der Maaten and Hinton (2008) and scikit learn (2020c) recommending that another dimensionality reduction algorithm be used initially to reduce the number of dimensions to a reasonable amount, for example, 50, if the number of features is very high.

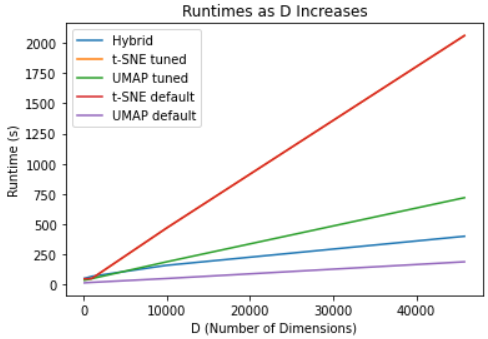


Figure 5.6: *Effect on runtime using default and chosen parameter values for t-SNE and UMAP with the best Spring Model (Hybrid Layout) runtime as a baseline.*

5.5 Summary

This section discussed the setup of the evaluation in order to create a fair environment. This included the hardware to be used and the independent, dependent and control variables involved in the evaluation. The set of parameters for each algorithm was also discussed including their chosen values.

The performance metrics of the evaluation were examined individually separating visual accuracy, runtime and memory usage. The latter two metrics each involved two separate approaches in order to evaluate each independent variable, N and D, and their effect. Random Projection was used as a baseline for visual accuracy and cluster identification and its impressive runtimes outperformed all other algorithms. However this was attributed to a major trade off for visual accuracy and so it was decided that this was unfair to be included in results. UMAP was found to be the next best performer in terms of runtime, providing the correct parameters were chosen and t-SNE proved to be the most memory efficient out of the six main algorithms; Random Projection, Chalmers' 1996, Hybrid Layout, Pivot Layout, t-SNE and UMAP. However there seems to be a large tradeoff for this low memory usage for runtime, especially as the size of the dataset increases. Chalmers' 1996 algorithm was found to be the worst performer for both runtime and memory usage but this was expected due to the logic, complexity of $O(N^2)$, and the caching techniques involved, requiring greater memory.

t-SNE and UMAP algorithms were further examined separately including comparing default parameter values to the ones chosen as well as how their metrics change using extremely large values of D. It was found that t-SNE struggled to finish in an efficient time compared to UMAP although this is a known drawback in the logical design of the algorithm which restricts its ability to create efficient layouts when extremely high dimensions are involved.

6 Conclusion

The aim of this project was to visualise iOS app usage data in a meaningful and informative way through the means of a careful process which involved comparing various dimensionality reduction algorithms. These algorithms included three Spring Models; Chalmers' 1996, Hybrid Layout and Pivot Layout algorithms as well as Random Projection, t-SNE and UMAP (Section 2.5). The idea was to conduct an evaluation comparing various performance metrics for each algorithm and present the results in a structured way.

This project was implemented in various stages each as important as the next. The iOS app usage data was pre-processed and analysed extensively before use with these algorithms to remove outliers, smooth noisy data and normalise values. Next, a collection of distance metrics were evaluated and compared with the aim to find the most appropriate one to best model the high-dimensional data. Clustering was carried out to identify potential groups of users, based on their usage, which would be visualised in the low-dimension. All six algorithms were successfully implemented. The three Spring Model algorithms were adapted from a previous student project (Cattermole 2019) developing these algorithms in Python. Each algorithm was run with the pre-processed data and appropriate visualisations were produced allowing ease of analysis through the coloring of nodes and interactability through node annotations.

An evaluation was performed on these visualisations with steps taken to ensure fairness throughout the process from hardware to parameter selection. Three performance metrics were evaluated individually; visual accuracy, runtime and memory usage. Each algorithm expectedly produced a different layout and so a subjective comparison was required for visual accuracy. Taking all factors into account, it could be shown that UMAP overall produced the most appropriate visualisation identifying global and local structure within the data better than any other algorithm although not without its downsides of struggling to preserve high-dimensional clusters. UMAP was the best performer in terms of runtime, however only on the omittance of Random Projection whose outperforming speeds were caused by a major tradeoff between runtime and visual accuracy. Although it was shown that UMAP was the most efficient only when specific parameters were chosen, proving the need for careful parameter selection and value assignment. t-SNE was the best performer for memory usage and this was consistent throughout all comparisons of the evaluation although this was also the reason the algorithm struggled to create layouts with a large number of data points efficiently.

Although not as recent developments as t-SNE and UMAP, Spring Models still performed well overall, in particular Hybrid and Pivot Layout algorithms. They registered roughly the same runtimes and although the complexity of Pivot Layout is lower, this was potentially caused through there only being 10,368 data points. Previous work has proved Pivot Layout to be more efficient on data points up to 1 million (Cattermole 2019). Memory usage was far greater however than that of t-SNE and UMAP however this was down to the logic of these algorithms where caching methods are involved unlike in t-SNE and UMAP where more memory efficient techniques are used. It is suggested that future work be carried out with the same algorithms, performance metrics and overall process but a larger dataset to further analyse differences.

Overall, this project showed that any dataset, no matter how obscure or sparse, can be visualised in an effective way through the following of a step-by-step process. Careful analysis and thoughtful

decision making must be carried out however in order to tune this process. In this case, interesting discussion points were raised about both the data and the mentioned algorithms. It was shown that a collection of seemingly random 10,368 iOS users could be carefully classified, grouped into distinct clusters and visualised in relatively fast times, proving how the application of a thorough process can produce an effective and fair implementation and evaluation.

6.1 Reflection

This project presented numerous challenges throughout which contributed to my experience as a software developer and also as a researcher. Significant time and patience was required in the beginning to understand the background research for this project and has shown me the complexity involved in large scale data visualisation projects and dimensionality reduction algorithms. However a number of benefits arose from this project distilling in me the need to plan meticulously and question every decision. Implementing these algorithms has allowed me to improve my knowledge of both the Python programming language greatly and its use within the field of Data Science. The evaluation showed me the importance of implementing fair experiments, providing me with a better understanding of the steps required to do this. Overall this was a challenging but rewarding project covering many aspects which will undoubtedly be used again throughout my software development career.

6.2 Future Work

This project has raised a number of suggestions regarding improvements that could be made or areas for future research. This section covers these suggestions for future work related to the project.

6.2.1 Cluster Analysis

This project has created the tools required for creating visualisations of app usage data. Future work could use these layouts and analyse what they represent, in app usage terms, taking into account clusters and other features of the layouts.

6.2.2 Alternative Datasets

This project has provided a thorough process for the visualisation of high-dimensional data, using iOS app usage data. It would be interesting to apply this process to alternative datasets of potentially different formats to evaluate how it compares and ultimately whether this is a one process fits all scenario or if modifications are required.

6.2.3 Larger N Values

The iOS app usage dataset, although consisting of up to 45,788 dimensions, had only 10,368 data points. If it was possible to increase the the number of users then the evaluation could be repeated further in depth with potentially more conclusive results, if larger datasets were available.

6.2.4 Parameter Selection

Another possible area for future work would be to apply the experimentation process described in Section 4.4.2 to feature the full range of parameters available for each algorithm. Due to time constraints this was not possible during this project and so only the most important parameters were selected and tuned. However it would be interesting to analyse the effect of every parameter and whether this leads to improved overall performance.

A Appendices

A.1 Pre-processing

	appID	total_launches	user_count	average_launches
1	1	5	2	2.500000
2	2	52	1	52.000000
3	3	260	38	6.842105
4	4	1	1	1.000000
5	5	1	1	1.000000
...	...	...	...	...
45784	45784	194	6	32.333333
45785	45785	3	1	3.000000
45786	45786	9	4	2.250000
45787	45787	3	1	3.000000
45788	45788	1	1	1.000000

45788 rows × 4 columns

Figure A.1: *DataFrame created to represent each app and its general usage across all users.*

A.2 Clustering Layouts

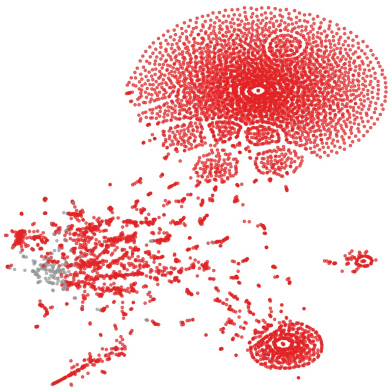


Figure A.2: *t-SNE layout showing 2 high-dimensional clusters.*

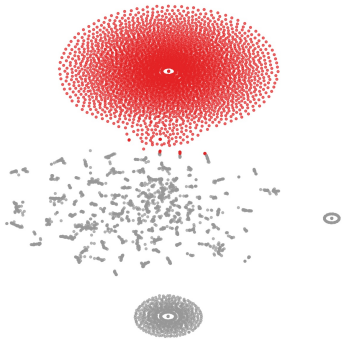


Figure A.3: *t-SNE layout showing 2 low-dimensional clusters.*

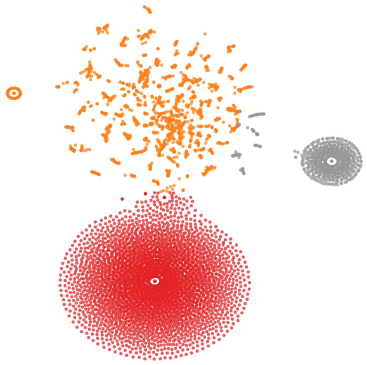


Figure A.4: *t-SNE layout showing 3 low-dimensional clusters.*

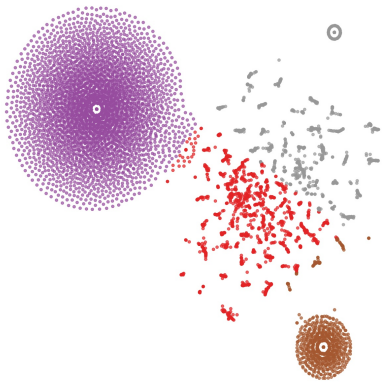


Figure A.5: *t-SNE layout showing 4 low-dimensional clusters.*

A.3 Progression Layouts

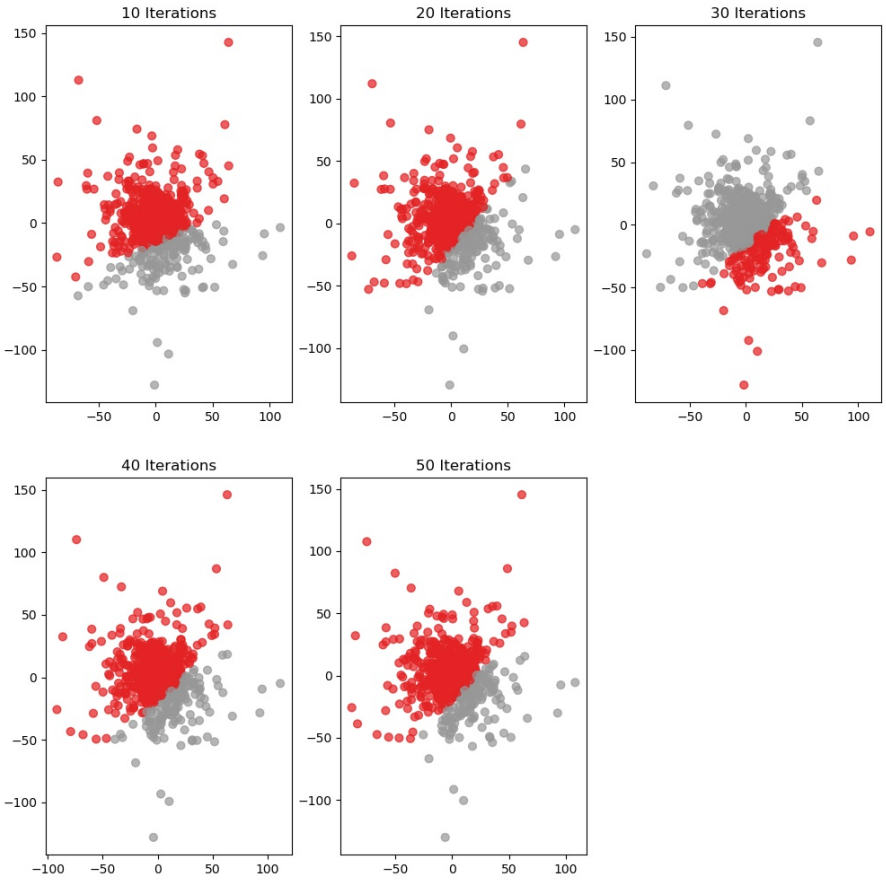


Figure A.6: *Chalmers96 algorithm showing its layout after every 10 iterations; 50 total iterations, showing how the algorithm refines positions of nodes as iterations progress.*

A.4 UMAP Layouts

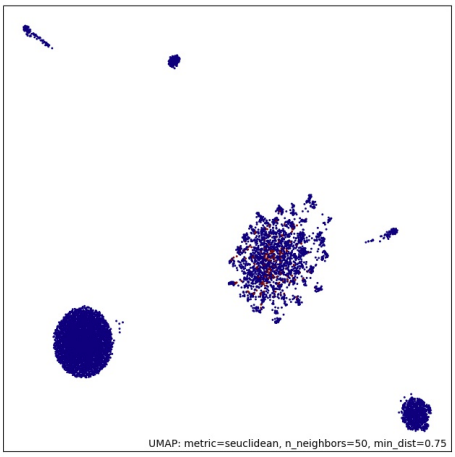


Figure A.7: *UMAP default layout.*

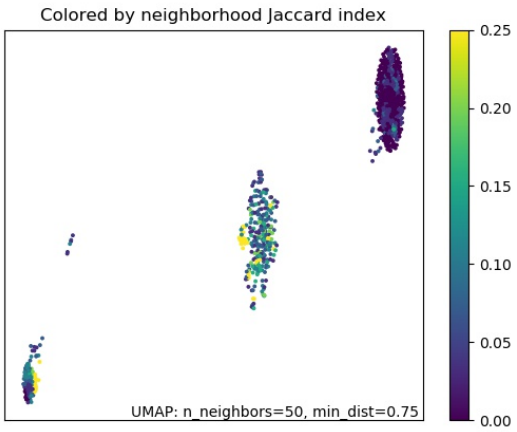


Figure A.8: *UMAP diagnostic layout.*

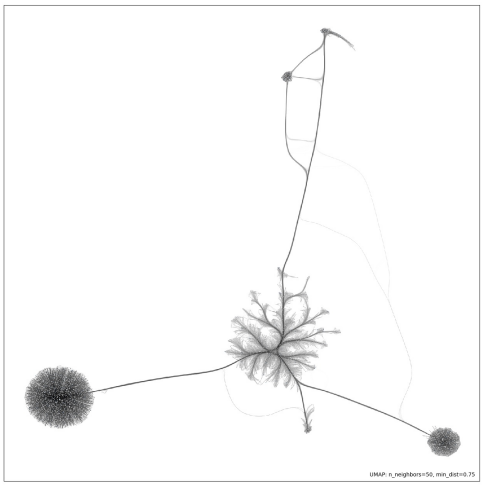


Figure A.9: *UMAP connectivity layout.*

Figure A.10: *UMAP first interactive layout*

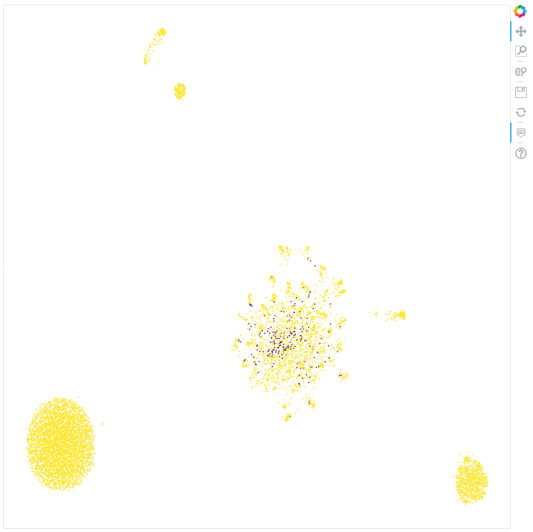


Figure A.11: *UMAP second interactive layout*

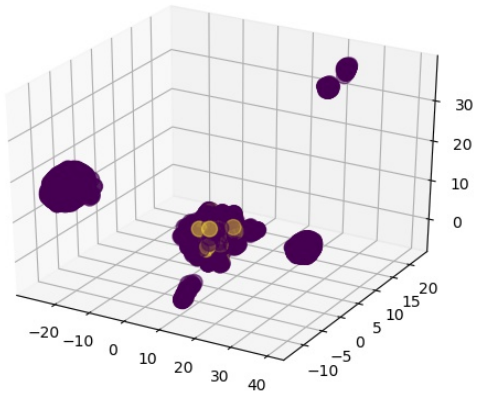


Figure A.12: *UMAP interactive 3D layout*

A.5 Runtime Results

Table A.1: *Runtime results where N was altered and D = 100.*

Algorithm	N (Users)				
	100	**500**	**1000**	**7504**	**10368**
Random Projection	0.003s	0.003s	0.003s	0.003s	0.004s
Chalmers' 1996	2.7s	14.6s	28.5s	222s	303s
Hybrid Layout	0.53s	2.9s	5.9s	53s	75s
Pivot Layout	0.66s	3.7s	7.5s	53s	73s
t-SNE	0.3s	1.8s	4.1s	46s	68s
UMAP	3.5s	6s	9.2s	38s	32s

Table A.2: *Runtime results where D was altered and N = 7504.*

Algorithm	D (Apps)				
	100	**500**	**1000**	**10000**	**45788**
Random Projection	0.003s	0.013s	0.024s	0.22s	0.95s
Chalmers' 1996	222s	254s	301s	645s	1643s
Hybrid Layout	53s	60s	70s	161s	401s
Pivot Layout	53s	63s	74s	160s	415s
t-SNE	46s	48s	51s	470s	2060s
UMAP	38s	43s	51s	160s	720s

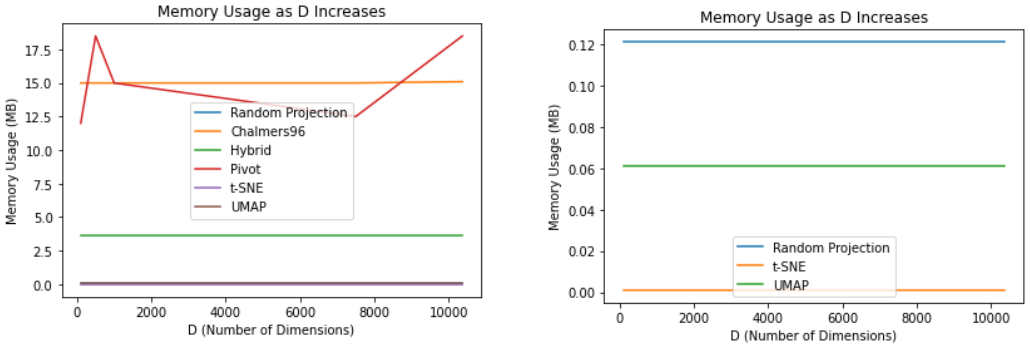
A.6 Memory Usage Results

Table A.3: *Memory usage results where N was altered and D = 100.*

Algorithm	N (Users)				
	100	**500**	**1000**	**7504**	**10368**
Random Projection	0.003 MB	0.009 MB	0.017 MB	0.121 MB	0.167 MB
Chalmers' 1996	0.168 MB	0.93 MB	1.19 MB	15 MB	19.68 MB
Hybrid Layout	0.055 MB	0.233 MB	0.486 MB	3.63 MB	4.88 MB
Pivot Layout	0.12 MB	0.52 MB	0.98 MB	12 MB	19 MB
t-SNE	0.001 MB	0.001 MB	0.001 MB	0.001 MB	0.001 MB
UMAP	0.002 MB	0.005 MB	0.009 MB	0.061 MB	0.084 MB

Table A.4: *Memory usage results where D was altered and N = 7504.*

Algorithm	D (Apps)				
	100	**500**	**1000**	**10000**	**45788**
Random Projection	0.121 MB	0.121 MB	0.121 MB	0.121 MB	0.121 MB
Chalmers' 1996	15 MB	15 MB	15 MB	15 MB	15 MB
Hybrid Layout	3.63 MB	3.63 MB	3.63 MB	3.63 MB	3.63 MB
Pivot Layout	12 MB	18.5 MB	15 MB	12.5 MB	18.5 MB
t-SNE	0.001 MB	0.001 MB	0.001 MB	0.001 MB	0.001 MB
UMAP	0.061 MB	0.061 MB	0.061 MB	0.061 MB	0.061 MB



(a) All algorithms. *(b) Only Random Projection, t-SNE and UMAP.*

Figure A.13: *Effect on memory usage as the number of dimensions/columns increases.*

6 Bibliography

R. Bartasius. Fast force-directed layout algorithms for the d3 visualisation toolkit. *School of Computing Science Honours Individual Project Dissertation*, 2017.

E. Bingham and H. Mannila. Random projection in dimensionality reduction: Applications to image and text data. *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 07 2001. doi: 10.1145/502512.502546. Date accessed: 30 March 2020.

M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, page 47–56, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305419. doi: 10.1145/2037373.2037383. URL https://doi.org/10.1145/2037373.2037383. Date accessed: 30 March 2020.

P. Boonsarngsuk. Faster force-directed layout algorithms for the d3 visualisation toolkit. *School of Computing Science Honours Individual Project Dissertation*, 2018.

A. Butt. 101 mobile marketing statistics and trends for 2020, 2020. URL <https://quoracreative.com/article/mobile-marketing-statistics>. Date accessed: 30 March 2020.

I. Cattermole. Force-directed layout algorithms for python. *School of Computing Science Honours Individual Project Dissertation*, 2019.

M. Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proceedings of Seventh Annual IEEE Visualization '96*. URL https://ieeexplore.ieee.org/document/567787.

DeepAI. Curse of dimensionality, May 2019a. URL https://deepai.org/machine-learning-glossary-and-terms/curse-of-dimensionality. Date accessed: 30 March 2020.

DeepAI. Jaccard index, May 2019b. URL https://deepai.org/machine-learning-glossary-and-terms/jaccard-index. Date accessed: 30 March 2020.

P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. URL https://ci.nii.ac.jp/naid/10000023432/en/. Date accessed: 30 March 2020.

EliteDataScience. Dimensionality reduction algorithms: Strengths and weaknesses, Jan 2019. URL https://elitedatascience.com/dimensionality-reduction-algorithms. Date accessed: 30 March 2020.

S. Kobourov. Spring embedders and force directed graph drawing algorithms. 01 2012. URL https://www.researchgate.net/publication/51989467_Spring_Embedders_and_Force_Directed_Graph_Drawing_Algorithms. Date accessed: 30 March 2020.

Matplotlib. About matplotlib. 2020. URL https://matplotlib.org/. Date accessed: 30 March 2020.

F. F. Mazda. *Telecommunications Engineers Reference Book*. Butterworth-Heinemann, 1993. Date accessed: 30 March 2020.

L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018. URL https://arxiv.org/abs/1802.03426. Date accessed: 30 March 2020.

A. Morrison and M. Chalmers. Improving hybrid mds with pivot-based searching. In *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No.03TH8714)*, pages 85–90, 2003. URL https://ieeexplore.ieee.org/document/1249012. Date accessed: 30 March 2020.

A. Morrison, G. Ross, and M. Chalmers. A hybrid layout algorithm for sub-quadratic multidimensional scaling. In *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pages 152–158, 2002. URL https://ieeexplore.ieee.org/document/1173161. Date accessed: 30 March 2020.

A. Morrison, X. Xiong, M. Higgs, M. Bell, and M. Chalmers. A large-scale study of iphone app launch behaviour. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3173918. URL https://doi-org.ezproxy.lib.gla.ac.uk/10.1145/3173574.3173918. Date accessed: 30 March 2020.

M. Nabil. Random projection and its applications, 2017. URL https://arxiv.org/abs/1710.03163. Date accessed: 30 March 2020.

neo4j. The cosine similarity algorithm, 2020. URL https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/cosine/. Date accessed: 30 March 2020.

L. H. Nguyen and S. Holmes. Ten quick tips for effective dimensionality reduction. *PLOS Computational Biology*, 15(6):1–19, 06 2019. doi: 10.1371/journal.pcbi.1006907. URL https://doi.org/10.1371/journal.pcbi.1006907. Date accessed: 30 March 2020.

NumPy. About numpy. 2020. URL https://numpy.org/. Date accessed: 30 March 2020.

pandas. pandas dataframe. 2020a. URL https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html. Date accessed: 30 March 2020.

pandas. About pandas. 2020b. URL https://pandas.pydata.org/about/index.html. Date accessed: 30 March 2020.

pandas. pandas series. 2020c. URL https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html#pandas.Series. Date accessed: 30 March 2020.

K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. doi: 10.1080/14786440109462720. URL https://doi.org/10.1080/14786440109462720. Date accessed: 30 March 2020.

Python Software Foundation. Python 3.7.7 documentation. 2018. URL https://docs.python.org/3.7/. Date accessed: 30 March 2020.

H. S. Ranjitkar and S. Karki. Comparison of a*, euclidean and manhattan distance using influence map in ms. pac-man. 2016. URL https://www.semanticscholar.org/paper/Comparison-of-A*%2C-Euclidean-and-Manhattan-distance-Ranjitkar-Karki/5688250a1f4449f0e6df06f83c635152864f6ba8. Date accessed: 30 March 2020.

scikit learn. Random projection. 2020a. URL https://scikit-learn.org/stable/modules/random_projection.html#random-projection. Date accessed: 30 March 2020.

scikit learn. About scikit-learn. 2020b. URL https://scikit-learn.org/stable/. Date accessed: 30 March 2020.

scikit learn. sklearn.manifold.tsne. 2020c. URL https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html#examples-using-sklearn-manifold-tsne. Date accessed: 30 March 2020.

SciPy. scipy.spatial.distance.pdist, 2019. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html. Date accessed: 30 March 2020.

SciPy. About scipy. 2020. URL https://www.scipy.org/about.html. Date accessed: 30 March 2020.

Seaborn. About seaborn. 2020. URL https://seaborn.pydata.org/. Date accessed: 30 March 2020.

F. Szabo. *The Linear Algebra Survival Guide: Illustrated with Mathematica*. Elsevier, 2015. Date accessed: 30 March 2020.

UMAP-Learn. Uniform manifold approximation and projection for dimension reduction, 2018. URL https://umap-learn.readthedocs.io/en/latest/. Date accessed: 30 March 2020.

UMAP-Learn. Umap api guide, 2020. URL https://umap-learn.readthedocs.io/en/latest/api.html. Date accessed: 30 March 2020.

L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL http://www.jmlr.org/papers/v9/vandermaaten08a.html. Date accessed: 30 March 2020.

M. Vlachos. *Dimensionality Reduction*, pages 274–279. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_216. URL https://doi.org/10.1007/978-0-387-30164-8_216. Date accessed: 30 March 2020.

M. Wattenberg, F. Viégas, and I. Johnson. How to use t-sne effectively, Oct 2016. URL https://distill.pub/2016/misread-tsne/. Date accessed: 30 March 2020.

P. Welke, I. Andone, K. Blaszkiewicz, and A. Markowetz. Differentiating smartphone users by app usage. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 519–523, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344616. doi: 10.1145/2971648.2971707. URL https://doi-org.ezproxy.lib.gla.ac.uk/10.1145/2971648.2971707. Date accessed: 30 March 2020.

L. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(65)90241-X. URL http://www.sciencedirect.com/science/article/pii/S001999586590241X. Date accessed: 06 April 2020.