IT UNIVERSITY OF COPENHAGEN

Clustering Player Behavior in Data Streams using MapReduce

by

Sigurdur Karl Magnusson

A thesis submitted in partial fulfillment for the
degree of Master of Science in IT

in the
Computer Science
Software Development and Technology

Supervisors
Julian Togelius, Rasmus Pagh

June 2013

"I know not with what weapons World War III will be fought, but World War IV will be fought with sticks and stones."

Albert Einstein

Abstract

Data is everywhere and it gets larger and more complex everyday. Mobile phones, internet activities, such as in social networks and online games are huge contributors to the data around us. There is a need for more efficient and scalable data analyzing algorithms to find patterns and connections in this pile of data. Especially in the recent years, where rise of Free-to-Play games, easily have millions of users, drive their revenue through micro-transactions and use player behavioural analysis to learn better about their customers. In this thesis a clustering algorithm called MapReduce k-means is implemented that incrementally clusters multiple batches of real life game data. The algorithm successfully finds general behaviour profiles, described by behavioural features from the game data. Results show that MR k-means finds stable and quality clusters, is efficient and scales when the data increases.

Acknowledgements

This thesis could not have been done without my wife Ingunn! Big thanks to her for all the understanding and being such a good wife. I also want to thank my parents (Magnus and Sigurros), my parents-in-law (Gretar and Gunnvor) and my brother-in-law (Sverrir) for their support.

Special thanks goes to Christian Thurau and Alaina Jensen

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AA	**A**rchetype **A**nalysis
EMR	**E**lastic **M**ap **R**educe
F2P	**F**ree **T**o **P**lay
GA	**G**ame **A**nalytics
GFS	**G**oogle **F**ile **S**ystem
MR	**M**ap**R**educe
MMO	**M**assive **M**ultiplayer **O**nline
RPG	**R**ole **P**lay **G**ame
SVIM	**S**implex **V**olume **M**aximization
SaaS	**S**oftware as a **S**ervice
SSE	**S**um of **S**quared **E**rror
ZMN	**Z**ero **M**ean **N**ormalization

Chapter 1

Introduction

1.1 Motivation

We live in a world where data is being generated at an amazing rate everywhere around us. Data can describe characteristics of e.g. Internet activities, social interactions, user behaviour in games, mobile phone activities, scientific experiments and measurements from different devices and sensor equipments. The amount of data that is being registered and stored around us is growing massively in volume and complexity. Organizations are storing more and more historic data than before, moving from large databases towards more commonly online distributed file systems or storage web services (e.g. Hadoop, Google, Amazon) providing scalability and high availability at commodity costs where *petabytes* ($10^{15} = 1.000$ *terabytes*) of information can be stored. We live in a world of *Big Data*, exploring unknown patterns and structures without knowing where they will lead us in the future.

"Information is the oil of the 21st century, and analytics is the combustion engine."

Peter Sondergaard, senior VP at Gartner

In digital games, data about in-game user interactions have been logged and behaviour analyzed since the first game was created. Analysis of the user experience and player behaviour have mostly been done in laboratories in the past, both during game development and after game launch to see if the game was played as designed. Game designs

have become increasingly complex in the recent years, offering much more freedom to the players by increasing the number of actions available, items to interact with and Massively Multi-player Online (MMO) persistent worlds that continue to exist after a player exits a game [3, 4]. This complexity generates much more user-centric data than before and is increasingly challenging when evaluating game designs [5, 6]. The user interactions being registered are called *user telemetry* which is translated to *game metrics* as referred to in game development, providing detailed and objective numbers, e.g. total playtime, monsters killed, puzzles solved.

Collecting user's telemetry can give very detailed quantitative information on player behaviour, and using data mining techniques can supplement traditional qualitative approaches with large-scale behavioural analysis [7], for example show where users are getting stuck and finding actionable behavioural profiles [3, 4, 8]. In the recent years, user behaviour analysis have in part been driven by the emergence of MMO games and Free-to-Play (F2P) games which can have millions of users and objects, which can form highly complex interactions. These game models, especially of persistent nature, are monitoring users actions and their behaviours to drive their revenue with subscriptions or offer players the opportunity to buy virtual items via micro transactions [3, 4, 6, 9].

One way of doing a behavioural analysis is to use an unsupervised machine learning technique called clustering. Cluster analysis is a popular exploratory data mining technique that groups sets of data objects together in a cluster that are more similar to each other than the data objects in other groups [10]. Human beings categorize or classify a new object or a phenomenon based on similarity or dissimilarity of the object's descriptive features and this is one of most primitive activities of humans [11]. Clustering explores the unknown patterns of the data and provides compressed data representation for large-scale data. In computer games, cluster analysis or behavioural categorization can find behavioural profiles that are actionable and give high valuable insights into the game development as well as increasing the monetization [12, 13].

Many clustering algorithms are designed for modern sizes of datasets where the whole dataset can fit into memory or allowing few passes (accesses) into a database (where each data object is read more than once). It can be very expensive to analyze large-scale datasets, and to get answers efficiently, one needs to reduce the set of data to

be analyzed, e.g. sample fewer players and have fewer features (dimensions) to be compared. Computations for large-scale data take time and need to be distributed to be able to complete in reasonable amount of time. Google's MapReduce programming model was introduced in 2004 [14] and allows automatic parallelization and distribution of computations on large clusters of commodity computers. Allowing programmers and researchers to easily implement highly scalable algorithms to process large amounts of data using the MapReduce model without worrying about handling failures and distributing the data with a large amount of complex code.

1.2 Problem Statement

> How can incremental clustering find general player behaviour in multiple batches of behavioural game metric data in a reasonable time?

Considering the massive size of user telemetry data being logged and processed, and the complexity of game designs. There is a knowledge gap when it comes to analyzing such large-scale data efficiently. Number of players are increasing and the complexity of player-game and player-player interactions grows exponentially. The largest massively multiplayer online role-playing game (MMORPG) *World of Warcraft* [1] had a population around of 10 million users (in 2012 from MMOData [2]), where players live in a persistent world that can create many millions of different and complex interactions in the game.

User telemetry from games can arrive in daily chunks and need to be processed incrementally (in mini batches). There is a need for algorithms that can process massive amount of data that doesn't fit in a computer memory to extracts knowledge in a reasonable time.

The goal with this project is to implement a scalable clustering algorithm to find the general behaviour in a specific real life game dataset in collaboration with GameAnalytics (GA) [16]. GA is a Software as a Service (SaaS) start-up, a data and analytics engine for game studios with its headquarters located in Copenhagen, analyzing large quantities of game metric data that need to be processed efficiently, returning actionable results to aid game design and development.

[1] http://en.wikipedia.org/wiki/World_of_Warcraft
[2] http://mmodata.blogspot.com

The successful outcome of this project depends upon:

- An efficient and scalable MapReduce clustering algorithm finding centroids in clusters describing the general behaviour of a real life game dataset provided by GA.
- The algorithm must be able to cluster multiple data batches incrementally and efficiently, where data chunks of game metric data arrive daily.
- The algorithm must find high quality centroids between multiple data batches.
- The general behaviour found must be intuitively interpretable and actionable to game developers.

1.3 Method

A k-means clustering algorithm in the MapReduce framework was implemented, enabling the high scalability and fault tolerance offered by MapReduce. Also a k-means start-up program was implemented that processes each data batch individually, running n-iterations of the MapReduce k-means clustering algorithm, which in the end returns the centroids for the clusters found in the data batch. These centroids are then used as input for the next clustering process, when the next data batch arrives. The MR k-means algorithm finds the nearest centroid for each point (player behaviour feature vector) in a *Map* function in MR and assigns it to the nearest centroid, then a *Reducer* MR function updates the centroids such that they they represent the current general behaviour.

Different MapReduce strategies were tested, e.g. implementing the *Combiner* function that is executed straight after the *Map* function has finished, minimizing the data transfer inside the MapReduce framework. Different methods in computing the distances from player behaviour data vectors to array of centroids, to find the nearest one centroid was implemented and tested. Such as the naive solution of processing each point and computing the distance by iterating over all the centroids and then over all the features/attributes and calculating the error. The other computing strategies were implemented and tested, using vector and array operations and also finding a distance matrix for all the points and the centroids by holding all the points in memory in the Mapper phase.

As recommended by GA, the *Python* programming language was used for the implementation under the 64-bit Linux (Ubuntu virtual box) operating system, ensuring compatibility with various libraries. The open-source Python MapReduce development framework *mrjob* was used and allows the programmer to easily implement the Map/Combine and Reduce functions needed. Mrjob enables the MR jobs to be easily run locally and in the cloud using the Amazon EMR web service. Python *vectorisation* libraries, such as *NumPy/SciPy*, were used to increase computing efficiency by groupping element-wise operations when manipulating data arrays.

1.4 Contributions

A MapReduce k-means clustering algorithm that incrementally processes multiple batches of data that e.g. arrive daily in chunks. The algorithm uses centroids from the previous data batch as input for the next data batch. Allowing the centroids of the data population seen before to adjust/move the centroids slowly to the current data batch without loosing too much of the effect from historic data.

The algorithm was evaluate using a real game data set from GA, where the clusters quality was measured after each data batch. Different number of clustering iterations on each data batch were compared, to see how much the cluster quality worsens between data batches or how stable it is. The multiple data batches are sorted after time and the oldest file is processed first, simulating the processing of data that arrive in chunks consecutively.

The MR k-means algorithm is also evaluated on a generated fictive data where three distributions of normal distributions or clusters move over multiple data batches, creating shifting centroids in the data. General player behaviours were interpreted from feature vectors (centroids) found in the game dataset. Various scalability (scale-out and scale-up) and efficient computations (computing the nearest centroid) were also evaluated using Amazon EMR.

The main results in this thesis shows that the MR k-means clustering algorithm finds general player behaviour by incrementally cluster multiple batches of real game data, and finds good quality clusters when performing low number of iterations on each data

batch. The results also show that the algorithm is efficient and scalable, and can easily be run using the Amazon EMR web service.

1.5 Project Outline

References are cited by index in the bibliography and are in order of appearance, e.g. [2] is a citation number two that is referenced in the thesis. Referring to other sections is by the number of that section, e.g. 4.1.2.

The organization of the thesis is as follows:

- **Chapter 2 - Background** Describes a short background theory about clustering player behaviour and the MapReduce framework for large-scale data parallel processing.
- **Chapter 3 - Related Work** Overview of related and recent work regarding clustering player behaviour and large-scale data with k-means as focus.
- **Chapter 4 - Methodology** Design and implementation work is described; Description of the real game dataset and selection of features, the k-means algorithm in MapReduce and the experimental set-up.
- **Chapter 5 - Results** Experiment results and observations are explained.
- **Chapter 6 - Discussions** Discussions about the experiment results.
- **Chapter 7 - Conclusions** Conclusions are drawn from the study including future research.

Chapter 2

Background Theory

2.1 Player Behaviour

When a user plays a game there can be a high number of action combinations that can be performed that affects the user and his surroundings. Interacting with objects or other players differently over a different time period. These user actions (user telemetry) are registered in the game and play a crucial role for many game genres to analyze and find player behaviour and how they are evolving over time. Online game genres like social online games or Free-to-Play (F2P), in *Facebook* and *Google Play*, drive their revenue by e.g. analyzing player behaviour to create actionable knowledge to be able to offer players to buy certain in-game items and upgrades for real money [3, 4, 6, 9].

Games running online persistent worlds where a game continues even a user quits the game, use player behaviour analysis e.g. for balancing gameplay, in-game economy and game design. Major commercial games have been using behaviour analysis for many years to help game design, Massively Multi-Player Online Role Play Games (MMORPG) play a special interests where the goal is to increase engagement of players playing the game in persistent worlds to have as many active monthly subscribers [7, 17].

F2P and Multi-Player Online (MMO) games allow large groups of players to interact in real-time, many of them in virtual worlds that are always running, allowing emergence of social communities in-game. These types of games can be highly driven by game metrics in game development, analyzing behaviour for e.g. customer acquisition and retention, evolving player communities and monetization [18].

2.1.1 Game Player Metric

When a user is playing a game he can e.g. interact with many items and other users, move through the game environment and purchase items. These kind of user actions is called user telemetry in context of collecting raw measurements data of user interactions in the game that is transmitted remotely to a game server where data about all users are stored and analyzed.

Game player metrics are interpretable quantitative measures of one or more attributes from user telemetry, related either to revenue or players perspective [1]. The revenue perspective player metric can be, e.g. Daily Active Users, Average Revenue Per User or when analyzing user attrition (churn) rate (a measure of the number of leaving users over a specific period of time). In players perspective, metrics are calculated related to user actions in-game, e.g. playtime of a user, average score, average hit/miss ratio, average puzzles solved.

Game metrics can be variables (features) or more complex aggregates or calculated values. For example hit/miss ratio can be calculated by aggregating actions of a user when firing a weapon by sum multiple attributes like "number of hits" and "number of misses" to calculate the hit/miss ratio feature. Also calculating a metric as function of time, e.g. using the features "player id", "session length" and "monsters killed" to calculate the metric "monsters killed per minute" for each player [1].

An example of a feature selection in a work by Drachen et al. [8], analyzing user behaviour telemetry in a major commercial title called *Battlefield 2: Bad Company*, a first-person shooter game strategic and tactical elements. Of over 100 features to choose from the authors carefully selected 11 features to allow for evaluating the most important gameplay mechanics [12], relating to character performance, game features and playtime. Some of the features selected were:

- **Score:** Number of points scored in total.
- **Skill level:** Aggregated value of player skill.
- **Total playtime:** Sum of player's time in total.
- **Kill/Death ratio:** Number of kills the player has scored divided with number of his deaths.

- **Score per minute:** Average score per minute of play.

A sub-category of player metrics called *gameplay metrics*, measures of all in-game player behaviour and interactions, e.g. navigation, using items and abilities, trading, using the game interface buttons and even the system/objects responses to player actions. Customer metric is another sub-category of the player metrics that usually receives higher priority considering the revenue chain in game development where it covers aspects of the user as a customer, e.g. virtual item purchases and other interactions with the game company, metrics that are interesting to marketing and the management [1]. See Figure 2.1 showing a game metrics diagram with focus on the player metrics hierarchically structure.

Gameplay metrics are however the most important player metric when evaluating player experience, game designs and quality assurance [3, 4, 8], but are generally in low priority compared to the customer metrics [1]. There are huge amount of player behaviour

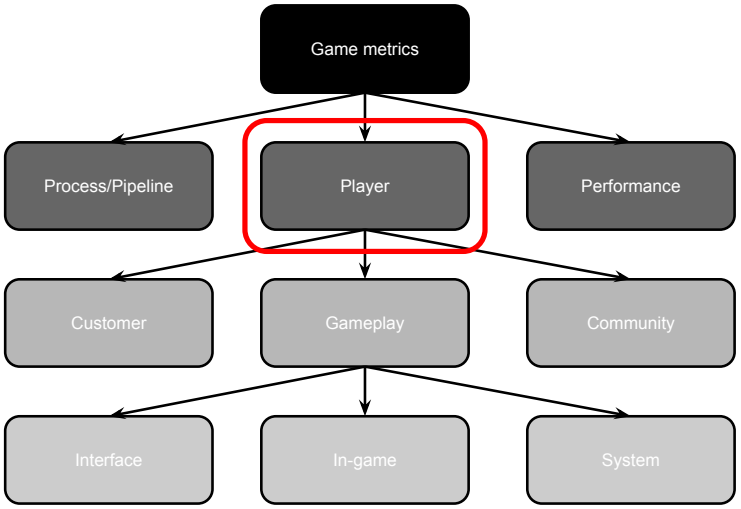


FIGURE 2.1: Player metrics hierarchical diagram. Figure taken from Drachen et al. *Game Analytics - The Basics* [1] and modified accordingly.

measures that are being logged each second in a complex game like MMORPG [3, 4], e.g. location of player, current health, stamina, status on weapons and a many more attributes that could describe a user state in a game also state/reactions of objects and monsters affected by the user. Its clear that this can generate a lot of information and game analysts face the problem of selecting the most essential pieces of information to analyze, imposing a bias but is necessary to avoid information overload [1].

2.2 Clustering

The goal of clustering is to categories or groups similar objects (players, items, behaviour or any observations) together into so called clusters (hidden data structures) while different objects belong to other clusters. A cluster is a set of objects that are similar to each other while objects in different clusters are dissimilar to each other, having high intra-cluster similarity and low inter-cluster similarity [19]. Identifying descriptive features of an object one can compare these features to a known object based on their similarity or dissimilarity based on some criteria. Cluster analysis can be achieved by various algorithms and is a common technique in statistical data analysis that is used in many fields, e.g. machine learning, pattern recognition, image analysis and bioinformatics. The main purposes of clustering has been used for following purposes

- Gaining insights into data, anomaly (outlier) detection and finding the most important describing features in the underlying structure.
- Identify the degree of similarity between objects, like natural classification.
- Compressing or organizing the data and summarize through cluster prototypes.

2.2.1 K-means clustering method

Many clustering methods exists but one of the most popular ones is called *k-means* [20, 21], also known as the Lloyd algorithm [22] which was further generalized for vector quantization by [19, 23, 24]. K-means seeks to group similar data points into k partitions or hyperspherical clusters giving insights into the general distributions in the dataset. A cluster is represented by a centroid that characterize the geometric center of the cluster that is calculated as the mean of all data instances belonging to that cluster. The objective function in k-means is to minimizes the squared error between a cluster's centroid (mean) and its assigned points and over all set of clusters minimizing the *Sum of Squared Error* (SSE) also called the *within-cluster variation*. Let a set of data points $x_i \in \Re^d, i = 1, ..., N$, where each point x is a real number d-dimensional vector and are partitioned into K clusters $C = \{c_1, ..., c_K\} \in \Re^d$, then the objective function is defined as

$$SSE = \sum_{k=1}^{K} \sum_{x_i \in c_k} dist(x_i, \mu_k)^2$$

where μ_k is the mean vector for the cluster centroid k and $dist(x_i, \mu_k)$ is a Euclidean distance measure between two points the data vector x_i and the mean vector μ_k. The mean vector μ_k is defined as

$$\mu_k = \frac{1}{N_k} \sum_{x_i \in c_k} x_i,$$

where c_k is a cluster number k and its N_k data points $i = 1, ..., N_k$. The Euclidean distance is the most popular distance measure for numeric attributes, also known as the *Minkowski distance* [1] a more generalized version using L_2 norm, the Euclidean distance function is defined as

$$dist(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

The above mathematical formulas were inspired from chapters 2. and 10. in the book *Data Mining: Concepts and Techniques* by Han et al. [25].

When running the k-means algorithm it starts by initializing K centroids by choosing random data points from the dataset or according some heuristic procedure. In each iteration of the algorithm it assigns data points to it's nearest centroid by calculating the minimum Euclidean distance to the K centroids for each instance. After assigning all the data points to clusters the centroids are updated so they represent the mean value of all the points in the corresponding cluster. The algorithm stops the iteration when the centroids do not change from the previous iteration or the error (SSE) is below some specific threshold. Also possible to manually define a maximum number of iterations to be run.

Algorithm 2.1 shows a pseudo-code of k-means. The algorithm can be seen as a gradient descent procedure which iteratively updates its initial cluster centers so as to decrease its error function.

[1] *http : //en.wikipedia.org/wiki/Minkowski_distance/*

Algorithm 2.1 K-MEANS(S, K)

Require: data set of instances $S = x_i, ..., x_n$. K number of clusters.
Ensure: K clusters

1: Initialize K cluster centers
2: **while** termination condition is not satisfied **do**
3: Assign instances to the closest cluster center
4: Update cluster centers based on the assignments
5: **end while**

One of the weaknesses of k-means is that it is sensitive of the initial selection of the centroids which can lead to local optimum, that is the algorithm converges and fails to find the global optimum. One solution is to run the algorithm n times and pick the initialization that gave the lowest SSE result. Another weakness of k-means is that a user has to predefine the number of centers k-means need to cluster, this is most often not known in advance. Many methods exist and most popular one is to run the algorithm with by increasing the k number of clusters to some K and pick a good k candidate where the *"elbow"* starts in the curve in a Scree plot [25]. Noise in data and outliers can dramatically increase the squared error and centroids shifting from data distribution in question towards outliers far away, thus representing skewed distributions. Solutions involve removing these noise in preprocessing or normalize the data with the zero mean normalization [10, 25].

The solution to the optimal partition can be found by checking all possibilities using a brute force method but that is a NP-hard problem [26] and cannot be solved in a reasonable time. The k-means algorithm is a heuristic approach for the clustering problem with running time of the algorithm $O(NKdT)$ where N is number data examples in d-dimensional space and T is the number of iterations. Usually K, d and T is much less than N meaning that k-means is good for clustering large-scale data because of approximately linear time complexity. The most computation work in k-means is the calculation of distances between objects and the cluster centers, in each iteration would require $O(Nk)$ distance computations. Calculating distances for each object is independent of each other, therefore these computations can be parallel executed.

The above implementation of k-means is called the *batch* k-means, where the centroids are updated after all the data points have been assigned. The *online* (incremental) mode of the algorithm processes each data point sequentially. For each data point x

the nearest cluster centroid c_{min} is calculated and that centroid is updated right away, defined as

$$c_{min}(old) = \underset{k}{\operatorname{argmin}} \|x - c_k\|$$

$$c_{min}(new) = c_{min}(old) + \eta(x - c_{min}(old))$$

where the cluster centroid c_{min} is updated towards the data point x using the learning rate η which determines the adaptation speed to each data point. The online approach is however highly dependent on the order of which the data points are processed. A variant of this method is used when clustering an endless stream of data where data points arrive one at a time or in chunks.

2.2.2 Clustering Player Behaviours

Cluster analysis in the context of game development is to find patterns of user behaviour, by reducing the dimensionality of the dataset in order to find the most important behavioural features and able to assign an expressive label to each found player profile. Behavioural clustering can answer questions like e.g. how people play the game, the average playing behaviour, evolving behaviour, finding extreme archetypes and find clusters that describe unwanted behaviour. Clustering player behaviour can give insights if a game is played as it is designed and obtain actionable knowledge that can be used to refine a game design [8].

There are many clustering methods that can be applied to find player behaviour, the most popular ones in recent years finding interpretable behaviour types (basis vectors or centroids) are *k-means* and Simplex Volume Maximization (SIVM) [2, 8]. Both clustering algorithms are applicable to large-scale data but are very different how they represent the behaviour types. Figure 2.2 shows an example of how different basis vectors can be found using SIVM and k-means.

SIVM is a Archetype Analysis (AA) approximation algorithm for large-scale data and its goal is to find the extreme (special) player behaviour whereas k-means focuses on the average player behaviour in compact clusters. Its easier to interpret the basis vectors from SIVM since they represent actual sparse data points where k-means cluster centroids (average of all data points in each cluster) can be very similar (in similar space)

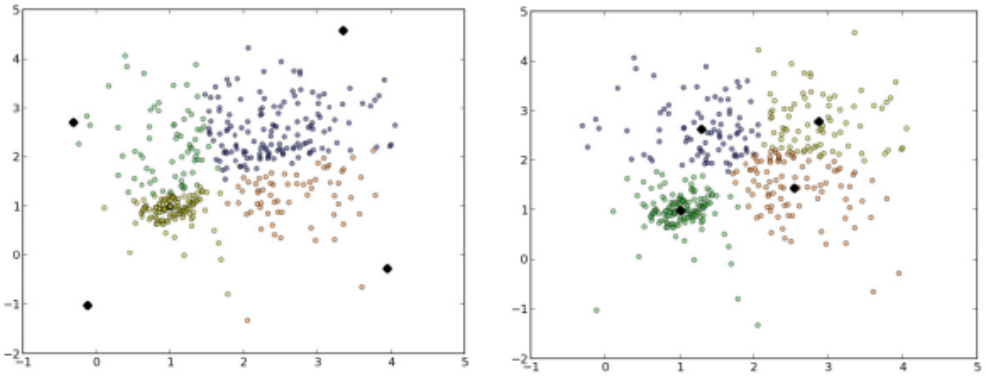


FIGURE 2.2: Simplex Volume Maximization (left picture) vs k-means. SIVM has four sparse basis vectors (black dots) and k-means has very similar centroids. The example is from a work of Drachen et al. [2]

and more difficult to assign an expressive label. There are can be many challenges that need to be addressed when applying clustering algorithms to game behaviour telemetry data e.g.

- The data can have a very high dimensionality, e.g. complex game interactions/mechanics from MMOGs.
- The need to mix many different behavioural data types and use of normalization methods, e.g. numeric, categorical and bi-nominal.
- Game telemetry data are often noisy and cleaning may be necessary, e.g. player pausing in a game, providing long play sessions.
- The need for defining parameters for clustering, e.g. number of clusters for k-means.
- Able to translate the findings to game developers, a knowledge that is actionable.

K-means algorithm has been shown to be very useful in behavioural analysis to obtain useful insights into in the general behaviour found in games, where AA like the SIVM algorithm can extract behaviour player profiles based on the different distances to each of the extreme behaviour [2]. When choosing features for behavioural cluster analysis, Drachen et al. [8, 27] suggests that one should focus on features related to the core game mechanics and playtime. Interpreting basis vectors found by SIVM and k-means are illustrated in Figure 2.3 and Figure 2.4, taken from a very recent study by Drachen et al. [2], authors selected playtime and leveling speed as behavioural variables showing

five leveling behaviour they found in the popular MMORPG game World of Warcraft. Each basis vector is a 2.555 dimensional vector where each entry represents the level of the player each day in the last 6 years.

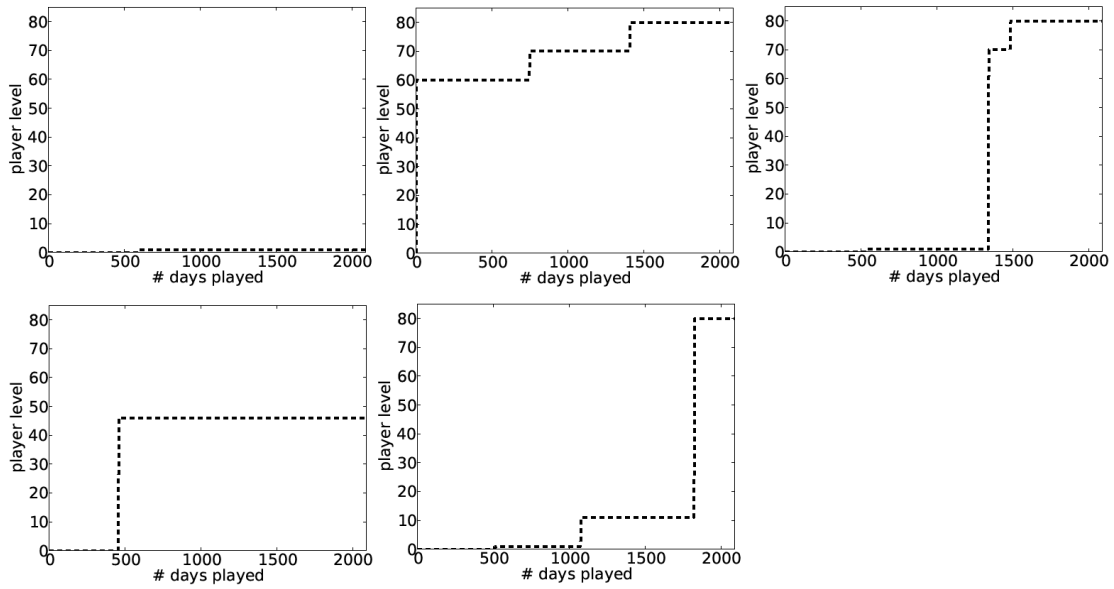


FIGURE 2.3: Five SIVM basis vectors showing extreme leveling behaviour for Archetype Analysis. Each basis vector is an actual player and is a legal playing behaviour thus very easy to interpret. This figure is from [2].

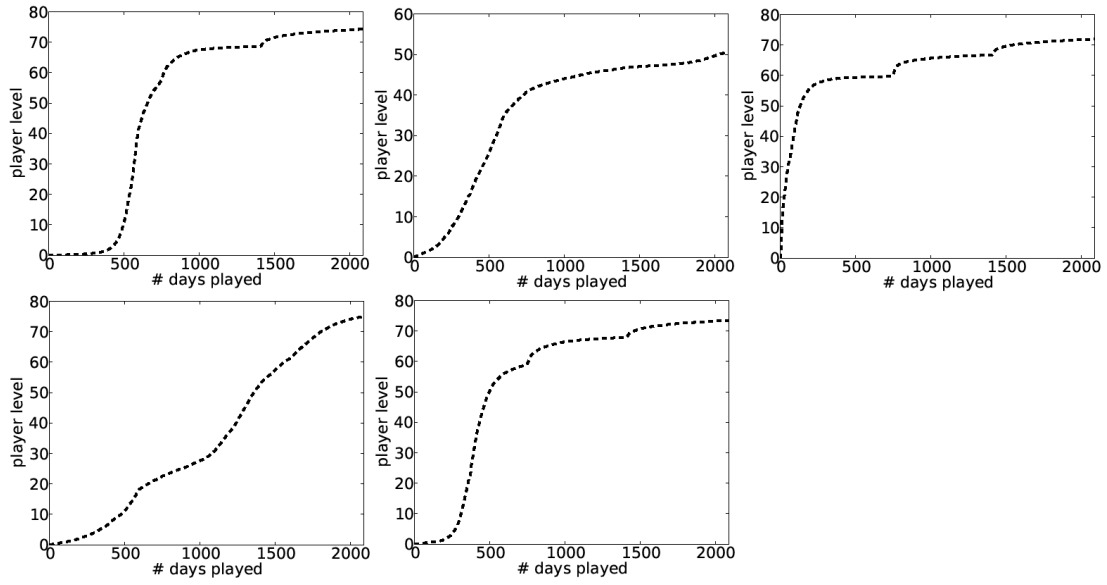


FIGURE 2.4: K-means five cluster centroids showing average leveling behaviour representing each cluster of players. They give good idea about leveling behaviour of many players but some of them are also very similar, interpretion is not straightforward. This figure is from [2].

2.3 MapReduce and Large-Scale Data

MapReduce is a programming model introduced by Google in 2004 [14], built on the divide-and-conquer paradigm, dividing a large-scale data into smaller chunks and process them in parallel. MapReduce enables fault-tolerant distributed computing on large-scale datasets and is a new way to interact with *Big Data* where as old techniques are more complicated, costly and time consuming [14, 28]. Google also introduced along with MapReduce a powerful distributed file system called *Google File System* (GFS) that could hold massive amount of data. This led to a new open source software framework called Hadoop [29], written in Java and is now maintained by Apache Foundation [2], a end-to-end solution for organizations that want to apply MapReduce. There are many Hadoop-related projects at Apache including the popular scalable machine learning and data mining library called Mahout, written also in Java.

Hadoop builds on the MapReduce and GFS foundation, designed to abstract away much of the complexity of distributed processing running on large clusters of commodity computers. The MapReduce programming model allows developers to write parallel distributed programs very easily by only implementing two functions called *Map* and *Reduce*. Developers don't need to worry about doing a complicated code to e.g. distribute work to computers, internal communication, data transfers and dealing with fault tolerance. Instead they can focus on the logic to solve the problem at a hand.

2.3.1 Programming Model

As mentioned before a developer only needs to implement two functions Map and Reduce. The *Map* function takes as input pair and produces an intermediate $(key, value)$ pair. The Map functions are run in parallel and produce many intermediate output pairs which is then grouped together with the same *key* by the MapReduce framework and is passed along to the *Reduce* function. The Reduce function receives a key and all of its set of values from all the Map functions, then it merges these values and typically produces zero or one output key and value pair per Reduce invocation.

Example: The problem of counting the number of occurrences of each word in a document, solved in MapReduce [14]. The Map function receives each word as input and

[2] http://www.apache.org/

emits the word as a key with count of 1, e.g. there is a set of words $w_1, ..., w_n$ in a document then the output from all Map functions would be the sequence of key value pairs: $(w_1, 1), ..., (w_n, 1)$. See the pseudo-code for the Map function in Algorithm 2.2.

Algorithm 2.2 MAP$(key, value)$

Require: key is document name, value is document contents.
Ensure: $(key, value)$ pair: (word, occurrence of one).

```
1: for all word ∈ value do
2:     EmitIntermediate(word, 1)
3: end for
```

The MapReduce framework then groups and merges all the $(key, value)$ pairs from all the Map functions and invokes the Reduce function with input key as a word and list of all the Map counts as the values: $(w_i, [v_1, ..., v_n])$ where w_i is a specific word with $1, ..., n$ counts of one, e.g. $(w_1, [1, 1, 1])$ if w_1 had three occurrences in the document. The reduce function will simply sum up all the counts and output a total count for each word, Algorithm 2.3 shows the pseudo-code for Reduce.

Algorithm 2.3 REDUCE$(key, values)$

Require: key is a word, values are list of occurrences for the word.
Ensure: number of occurrences for a word.

```
1: count ← 0
2: for all value ∈ values do
3:     count = count + value
4: end for
5: Emit(AsString(key, count))
```

The MapReduce also allows the developer to implement a *Combiner* function to do a partial reducing task that is executed on the same computer node as the Map function, combining the intermediate values after execution of the Map function. In the example mentioned above if a Combiner is implemented it will receive the Map output $(w_1, 1), ..., (w_n, 1)$ as input. The Combiner would then partly sum up the occurrences for each word (key) and output a intermediate sum for each word from the Map function. E.g. if w_1 had three occurrences then the output pair would be $(w_1, 3)$ instead of $(w_1, 1), (w_1, 1), (w_1, 1)$. The algorithm for a combine function can be found in Algorithm 2.4, it can look exactly like the Reduce function that was described before since it is combining/reducing the information.

Algorithm 2.4 COMBINE($key, values$)

Require: Key pair from Map, where key is a word and value is a occurrences of one.
Ensure: ($key, value$) pair: (word, partly sum of occurrences).

```
1: count ← 0
2: for all value ∈ values do
3:     count = count + value
4: end for
5: EmitIntermediate(key, count)
```

Using a Combiner reduces the information sent over the network to the Reduce functions by eliminating repetition of intermediate keys produced by the Map tasks. Input for the Reduce function is the same as before, a word and a list of all occurrences where they are now a partly sum values from all the Combiners, e.g. if w_1 has three occurrences from one Combiner and two occurrences from another, the input will be: $(w_1, [3, 2])$, instead of $(w_1, [1, 1, 1, 1, 1])$

2.3.2 MapReduce Hadoop Execution Flow

When a user program calls the MapReduce program the following sequence of occurs. MapReduce splits the input data into M pieces (typically 64MB) then starts up many copies of the user program on a cluster of machines. One of the copies is a *Master Controller*, the rest are workers which each get assigned a Map or a Reduce task from the master. The master keeps track of the M Map and R (user specified) Reduce tasks statuses, scheduling of tasks, re-executing work from failed workers and responsible for messaging and information flow between workers.

Each Map task is assigned one or more chunks of the input data and executes the Map function code written by the user. The Map task output is stored on a local disk in R partitions, the master then notifies the Reduce task about the intermediate data. The Reduce task reads the data and sorts it by the intermediate keys, grouping all occurrences of the same key together. For each unique intermediate key encountered it passes the key and the corresponding occurrences to the Reduce function written by the user. Figure 2.5 illustrates the execution flow that was described earlier, for more details one can look at work of Dean and Ghemawat [14] and Rajaraman and Ullman [30].

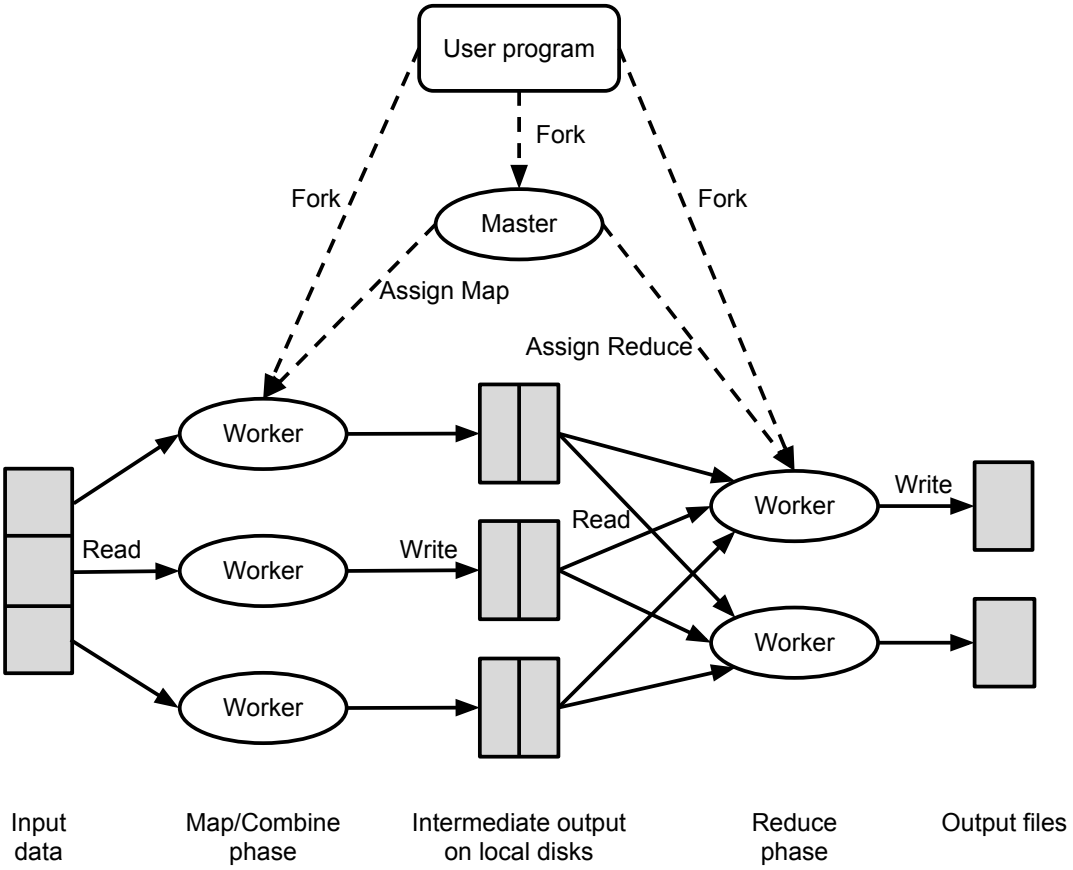


FIGURE 2.5: Overview of the MapReduce execution flow

Chapter 3

Related Work

Clustering player behaviour and processing large data sets have been researched actively in the recent years. In this thesis general player behaviour is found in a real life game data, which GameAnalytics contributes. Using k-means clustering algorithm to find k clusters representing average player behaviour, implemented in the MapReduce framework for high scalability and parallel processing for large-scale data. The behaviour of the data can change from day to day like when dealing with an endless evolving stream of data is discussed. In subsequent sections some of the related work and recent studies are given a short introduction.

3.1 Clustering Player Behaviours

Many researches have been done on clustering and predicting player behaviour over the last years to get a better understanding which kind of user behaviour are to be found when playing a game that can be actionable for game developers [4, 8, 31–33]. User behaviour analysis has becoming increasingly popular in the recent years because of rise of the Free-to-Play (F2P) genre games in *Facebook* and *Google Play* where populations can be in millions creating complex game interactions [3, 4]. Playing these games are free and many are of persistent nature where the world in the game continues when a player exits. To be profitable these games drive their revenue via micro transactions, e.g. players buying upgrades or virtual items in game for real money [3, 4, 6, 9]. Major game publishers have also been collecting and analyzing large scale of behaviour telemetry

data, e.g. maximizing player engagement to sell them monthly subscriptions, but details of their methods are kept confidential [7, 17]. Most available research work is case-based where a specific algorithm is applied to a specific game and commercial game data sets has only become accessible recently for academic researchers [7].

Predicting player behaviour in a major commercial game Tomb Raider: Underworld was presented in a study of Drachen et al. [12]. Authors classified 1365 players in a moderate data set into four user behavioural groups using six statistical gameplay features based on core game design as inputs of an emergent self-organizing map to identify dissimilar behaviour clusters. Behaviour profiles covering 90 percent of the users in the dataset were labeled in game terminology usable for game designers. Mahlmann et al. [13] did a follow up on the research using eight gameplay features and classified behaviour of 10,000 players. The authors presented also how to predict behaviour based on early play analysis, a popular topic which can be used to prevent attrition [9].

Analyzing social groups in the highly popular Massively Multiplayer Online Role-Playing Game (MMORPG) World of Warcraft was done by Thurau and Bauckhage [15]. They analyzed how groups (guilds) evolve over time from both American and European based guilds. Their paper is the first study analyzing such amount of data in a MMORPG, analyzing large-scale data gathered on-line from 18 million players belonging in 1.4 million groups over a period of 4 years. Convex-Hull Non Negative Matrix Factorization (CH-NMF) [33] technique was applied to the data to find the extremes rather than averages and results show no significant cultural difference in formation processes of guilds from either the US or the EU. Interpretability of CH-NMF was more distinguishable in representing archetypal guilds than the more conventional clustering method k-means that represent the cluster centroids with similar characteristic.

Drachen et al. [8] did a clustering analysis for two major commercial games applied to large-scale of high-dimensionality player behaviour telemetry. K-means and Simplex Volume Maximization (SIVM) clustering were applied to the MMORPG *Tera* and the multi-player first-person shooter strategy game *Battlefield 2: Bad Company 2*. SIVM clustering is an adaption of Archetype Analysis (AA) for large-scale data sets to find extreme player behaviour profiles [33, 34]. The authors show the contribution differences from the two algorithms where k-means gives insights into the general distribution of behaviour vs. SIVM showing players with extreme behaviour. The selection of the

most important features from the data set were followed by a method suggested by Drachen et al. [12], behavioural profiles were extracted and interpreted in terms of design language [12, 17].

In a recent study by Drachen et al. [2] the authors compare four different popular methods with purpose of clustering player behaviour and develop profiles from large-scale game metric data set from the highly popular commercial MMORPG World of Warcraft. The data set was collected from mining the Warcraft Realms site, recordings of on-line time and what level each player reached for each day in the years 2005-2010 for approx. 70 thousands of players, creating a 2.555 dimensional feature vector where each entry corresponds to the level of the player for each day. The authors selected playtime and leveling speed as their behavioural variables to show a measure of the overall player engagement in the game, where playtime is one of the most important measure for calculating the churn rate [6, 9]. Interpretable behaviour profiles where only generated by the k-means and the SIVM algorithm. The SIVM Archetype Analysis algorithm produces however significantly different behaviour that result in easier interpretation of behaviour profiles compared to the k-means algorithm where the centroids are overall similar.

3.2 Clustering Large Data

K-means [20–22] is one of the most studied clustering algorithm out there and is still actively researched. It's a simple algorithm that partition the data into k partitions by minimizing its objective function sum of squared error (SSE). From its appearance almost 50 years ago it has been one of the most popular clustering algorithm to research because its ease of interpretation and simplicity [10, 35, 36]. One of the problems with k-means is that it is a heuristic algorithm and has no guarantee to converge to a global optimum (optimal solution). Many work have been done in researching approximation guarantees (guarantee a approximated solution which is a within a constant-factor of the optimal solution) for k-means both in non-streaming and streaming versions [37–41]. In recent years k-means has also been very popular algorithm to study in the MapReduce framework where the algorithm can easily be applied to cluster large amount of data sets in parallel [14, 42–45].

Guha et al. [46] designed an algorithm in 2003 called STREAM that is based on the divide and conquer strategy to solve the k-median problem (a k-means variant), authors guarantee a approximated guaranteed solution. The algorithm divides the dataset into m pieces of similar sizes, where each of the pieces are independently clustered sequentially and all the centers from all the pieces are then clustered further. They show a new k-median algorithm called LSEARCH that is used by the STREAM algorithm and is based on a local search algorithm solving the facility location problem [47] to solve the k-median problem. Results show that STREAM LSEARCH produced near optimal quality clusters and better than STREAM k-means also with smaller variance. Compared to the the hierarchical algorithm BIRCH [48] it took 2-3 times longer to run but produced 2-3 times better quality clusters (SSE), showing superior strength when the goal is to minimize the SSE like detecting intrusions in networks [49].

In 2009 Ailon et al. [39] extended the the work of Guha et al. mentioned above by introducing a new single pass streaming algorithm for k-means, first of its kind with approximation guarantees. Achieving this they also designed a new algorithm called k-means# that is based on a randomized seeding procedure from the non-streaming algorithm k-means++ by Arthur and Vassilvitskii [38]. The k-means++ chooses k centers non-uniformly whereas k-means# selects $O(k \log k)$ centers and achieves a constant approximation guarantee. In the streaming algorithm they run the k-means# independently on each divided piece of the data to achieve $O(k \log k)$ random centers non-uniformly and use the k-means++ algorithm to find k centers from the intermediate centers from all the pieces of the data set.

Another approach using a *coreset* by selecting a weighted subset from the original dataset such that by running any k-means algorithm on the subset will give near similar results to running k-means on the whole dataset. Ackermann et al. [50] introduced a streaming algorithm called StreamKM++ that is a streaming version of k-means++ algorithm from Arthur et al. [38] to solve k-means on the weighted subset and a new data structure called coreset tree, speeding up the time for the sampling in the center initialization. Their approach was shown to produce similar quality of clusters (in terms of SSE) as the STREAM LSEARCH [46] algorithm but scaling much better with number of clusters centers.

Shindler et al. [41] proposed an algorithm called *Fast streaming k-means* based on the

online facility location algorithm [51] and extends the work of Braverman et al. [40]. Authors prove that their algorithm has a much faster running time and often better cluster quality than the divide and conquer algorithm introduced by Ailon et al. [39]. The algorithm however show similar average results as StreamKM++ [50] in both running time and quality tho with better accuracy.

Clustering data streams of an unknown length, evolving over time [52, 53] are challenging where it is not possible to access historic data points because of the amount of data arriving continuously. Aggarwal et al. [54] proposed a well-known stream clustering framework called CluStream for clustering large evolving data streams and is guided by application-centered requirements. CluStream has an online component that maintains snapshots of statistical information about micro-clusters (a.k.a. *cluster feature vector* [48]) in a pyramidal time window and an offline component that uses the compact intermediate summary statistics from the micro-clusters to find higher level k clusters using k-means, in a time horizon defined by an analyst. A new high-dimensional highly scalable data stream clustering algorithm called HPStream was also proposed [55]. HPStream uses projected clustering [56], which can determine clusters for a subset of dimensions, to data streams and a new data structure called *fading cluster structure* that allows historical and current data to integrate nicely with a user-specified fading factor.

Another approach clustering evolving streams Zhou et al. [57] presented a algorithm called SWClustering to cluster evolving data streams over so called sliding windows, where the most recent records are considered to be more critical than historic data [58]. Allowing to analyze the evolution of the individual clusters by eliminating influence by outdated historic data points when new data points arrive. Authors show that the CluStream algorithm [54] is much more sensitive to influences of outdated data and is less efficient.

Processing large of amount of data efficiently using parallel processing is an active research and gain much of popularity when the Google's MapReduce programming model was introduced by Dean and Ghemawat [14]. Zhao et al. [42] implemented a parallel version of k-means (PKMeans) in the MapReduce framework and showing their algorithm can be effectively run on large data sets. The Map function calculates the distance to the

closest cluster centroid for each data point at a time, after assigning to the clusters a combiner sums up all the data points dimensions for each cluster from that Map function and outputs the key and value pair $(clustercentroid, [sumforalldimensions, numberofpoints])$. The Reduce function then sums up all the intermediate sum values for each cluster and calculates the mean for the new centroids.

Li et al. [59] implemented the algorithm MBK-means in MapReduce using the Bagging ensemble learning method [60], using replacement sampling to generate k new data sets from the original data. K-means algorithm clusters each new data set using the MapReduce framework until convergence and in the end all the centroids from the k sets are merged to form the final k centroids.

Many extensions on the traditional MapReduce framework have been proposed to support efficient algorithms running iteratively [61–66] and incrementally [66–68]. The incremental MapReduce frameworks are interesting and relates to the work in this thesis, incrementally clustering chunks of data but are not under research in this thesis.

3.3 This study

The MapReduce k-means algorithm implementation in this thesis is most similar to PKMeans [42] described above, a parallel k-means implementation in MapReduce using a Combine function to reduce the intermediate data sent between the mappers and the reducers. In this thesis a parallel k-means algorithm in MapReduce is implemented so that each Map function efficiently calculates the distance to nearest cluster centers by calculating the distance between a data points and the cluster centers in a few array operations using *vectorisation*. The MR k-means algorithm is also applied to incrementally cluster multiple batches of player behaviour data batches both on a real game data and on a generated data, where the centroids are rapidly changing between sets of data.

In this work the incremental MapReduce k-means algorithm results as a good approach to provide valuable insights into the general behaviour for multiple batches of a real game data provided by GameAnalytics [16]. Work of Drachen et al. [2, 8] use k-means to cluster player behaviour and was a inspiration to this study such as interpretation of cluster centroids and approaches to select and build important game behavioural variables from user's telemetry and extracting behavioural profiles. Additional experiments

were performed with a generated data set having different normal distributions of data in multiple batches of data to simulate different distributions of data arriving daily.

Chapter 4

Methodology

In this chapter the real game dataset and the implementation of MapReduce k-means clustering algorithm is described. There is a section about preprocessing the data, constructing the player behaviour features and explaining the MR k-means implementation process and the MR k-means program that launches the MR algorithm in each iteration, incrementally clustering multiple batches of data.

In GA the user telemetry data arrives in mini batches from the games, a batch each day. In this thesis the idea is then to process each batch and incrementally cluster the data, where the output from one batch is the input to the next.

An incremental MR k-means clustering algorithm and apply it on multiple batches of the game metric data to find average player behaviour described by a the constructed player behavioural feature vector. The algorithm is highly scalable and can easily be executed on numerous virtual computers using the Amazon Elastic MapReduce (Amazon EMR) web service.

Different MR k-means implementation were tried in the process of this thesis and different efficient computation methods were compared on a larger generated dataset, including scalability (scale-up and scale-out) tests running on different cluster setups in the Amazon EMR. In the last section a quick overview of the development environment and tools that were used in this thesis is covered.

4.1 Dataset

The dataset is from a Free-to-Play (F2P) online game that is available on Google Plus and Facebook. This real life game dataset is from one of the many games that GameAnalytics is analyzing for their customers. Because of a confidentiality agreement its name is not mentioned in this thesis, but lets call it *Free Battle Online*. Same is about the real names of events or features in the dataset is fictional, this unfortunately means that there is a very limited knowledge about the data, e.g. what individual events mean and the timespan. The goal of the game is to fight battles against both non-playing-characters (NPCs) and other players. You can click on various places to travel to finish battle missions or just visit other players and initiate battles to steal their resources.

The dataset is a user telemetry data, e.g. user behavioural events, where each line represents an action performed by the user or is a direct influence from a user behaviour. There are over 5.100.000 rows in this data set, with approx. 550 unique events generated by approx. 94.000 players. Average events per player is $\mu = 54$ with standard deviation $\sigma = 145$, indicating that the data is spread out over a wide range of values. The measure of the spread is in Table 4.1 showing the five number summary; The first quartile Q_1 cuts off the lowest 25% of the data, the second quartile gives the center of the distributions known as the $Median$, the third quartile Q_3 cuts of the highest 25%. The median is lower than the mean of the data describing a positively skewed distribution, see Figure 4.1 for a visualization of the distribution.

	Min	Q_1	$Median$	Q_3	Max
Events	1	4	23	61	10865

TABLE 4.1: This table shows the five number summary about the number of events generated by each unique player in the dataset.

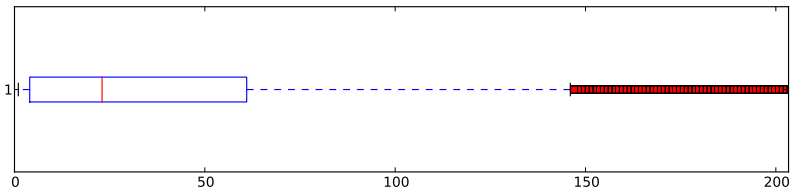


FIGURE 4.1: A boxplot visualizing the Events distribution incorporating the five number summary.

The two lines outside the box in Figure 4.1 are called whiskers and represent the extreme low and high values that are less than $1.5 \times IQR$ beyond the quartiles, IQR is an

interquartile range defined as $Q3 - Q1$ a range covered by the middle half of the data. The first and third quartiles represent the ends of the box and the median is the red line. The red *squares* are individual points called *outliers*. Only about 6% or ≈ 5500 players of the entire population generated more than 150 events, 0.3% or ≈ 300 players have more than 1000 events and less than ten people have a range of $5.000 - 10.000$ generated events! A histogram showing the frequency of events generated by players is shown in Figure 4.2, zooming in on the spread that gives the range covered by the middle half of the data.

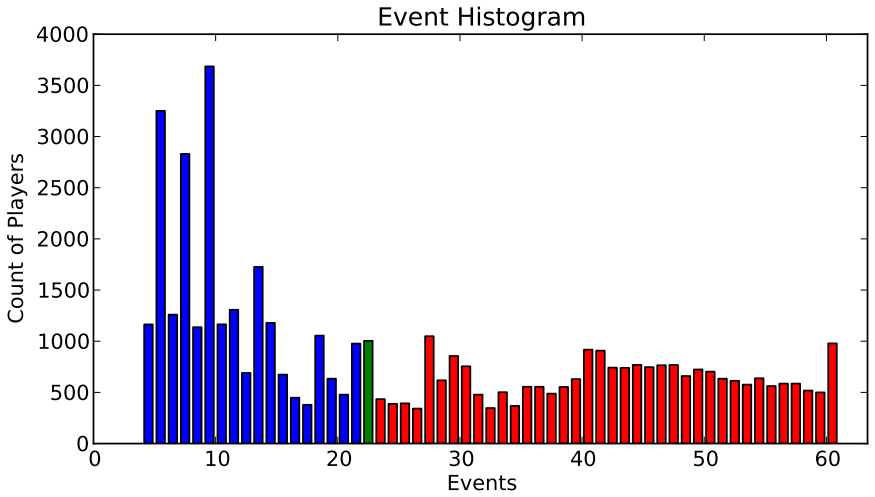


FIGURE 4.2: A histogram for number of events using singleton buckets in the interquartile range, defined as $Q3 - Q1$, range covered by the middle half of the data.

4.1.1 Game Metric Construction

For this thesis the number of game features were kept at minimal and it was decided to build the feature vector with three features. Given the limited knowledge and information that were available about individual events, it was decided to select events that had high frequency compared to other events and hopefully could describe different behaviour between groups of players in those features. A $3-dimensional$ feature vector was constructed for each unique player, containing a frequency of three events generated by the player. The aggregated events are:

- Login: Number of logins/access into different areas in the game($\mu = 2.4, \sigma = 4.8$)
- Battle: Number of battles this player have started ($\mu = 3.7, \sigma = 9.6$)

- Premium Spending: Number of times this player spends a in-game money on virtual items or resources ($\mu = 1.1, \sigma = 2.7$)

Features were chosen such that they could describe player engagement to the game. How active the player is exploring various areas, fighting battles and spending in-game money to advance in the game quicker, see Figure 4.3 how the data looks like in data space.

The five number summary of the skewed distribution is in Table 4.2. Boxplots visualizing the distributions for these events are shown in relevant figures below; Logins in Figure 4.4, Battles in Figure 4.5 and Premium spent events in Figure 4.6.

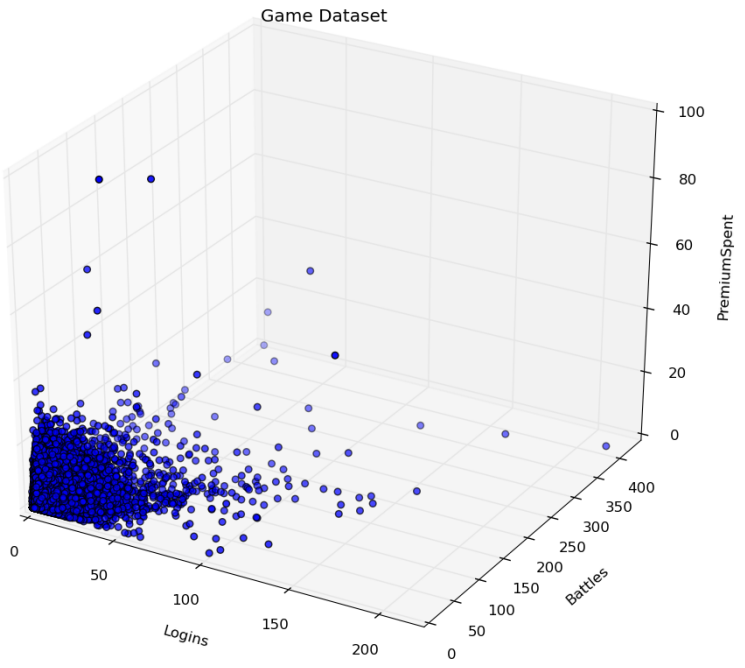


FIGURE 4.3: This figure shows the whole game metric dataset plotted in 3-dimensional data space. Showing the whole range for the data features.

	Min	Q_1	*Median*	Q_3	*Max*
Logins	0	1	1	2	220
Battles	0	0	1	4	439
Premium spent	0	0	0	1	100

TABLE 4.2: This table shows the five number summary for the Logins, Battles and Premium spent events generated by each unique player in the dataset.

The features values are spread out over large range of values, in the preprocessing Section 4.1.2 it is explained how the outliers or anomalies were detected and removed from the data before applying the MR k-means algorithm using a *z-score* method.

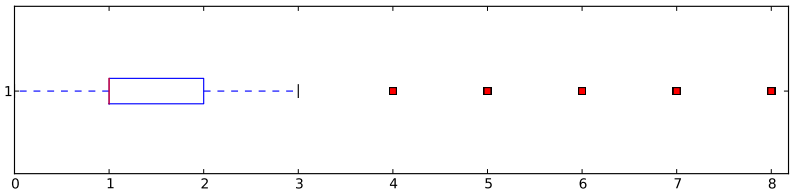


FIGURE 4.4: A boxplot visualizing the Login events distribution incorporating the five number summary.

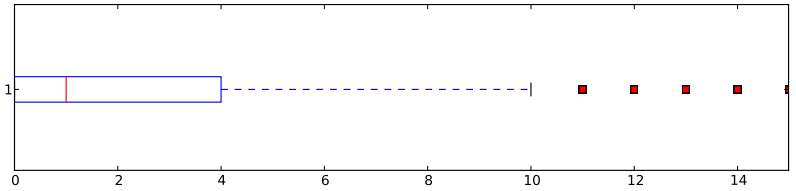


FIGURE 4.5: A boxplot visualizing the Battle events distribution incorporating the five number summary.

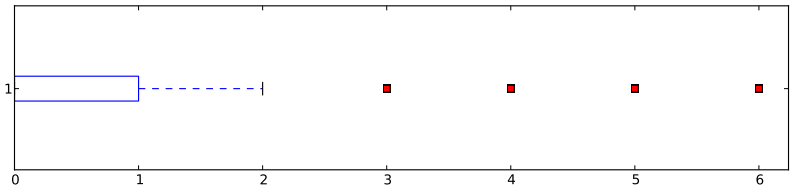


FIGURE 4.6: A boxplot visualizing the Premium spent events distribution incorporating the five number summary.

4.1.2 Preprocessing

User telemetry from games can be very noisy in that sense it can contain lots of irrelevant events and logging information and even developers from the game studio can generate events. For this dataset the following preprocessing steps were performed:

- Sort the events after event timestamp.
- Split the dataset into 10 approx. equally sized data set pieces.
- Filter out irrelevant events and logging information. E.g. Events not origin from Facebook or Google Plus, non related events and in game studio debugging generated events.
- For each data set piece:

- Counting the events mentioned above per user to build the feature vector.
- Remove outliers or anomalies from the data.
- Standardize the data to standard scores also called *z-score*. Transforming the data to standard normal distribution, with $\mu = 0$ and $\sigma = 1$.

To apply the incremental MR k-means algorithm it was needed to have multiple batches of data to process incrementally. The real game dataset from GA was used and split up into 10 equally sized data batches sorted after the event arrival timestamp. The idea was to try to simulate that the algorithm is processing multiple batches of data in the same timely order as the events were generated over multiple time-periods or days, like at GA. Next irrelevant information are filtered out, e.g. information that don't contribute as a event logging or are obviously from the game designers themselves.

Each data batch is processed by aggregating or counting the events that were selected before to build the feature vector for each unique player in the data set. Next the outliers were found by transforming the data to standard scores using the *zero mean normalization* (ZMN), also called *z-score*, with standard normal distribution having zero mean and unit variance $\mu = 0$ and $\sigma = 1$. A feature vector that has a feature value z-score that is more than 3 standard deviations away from the mean is considered an outlier in the data and is removed from the original data batch. Having a data batch without outliers, the data can be transformed again using ZMN, without having the effect from the outliers. The ZMN is defined as follows

$$z = \frac{x - \mu}{\sigma}$$

Using ZMN is widely used in machine learning algorithms and is needed when calculating the distance between two points (feature vectors) such that each feature contributes approximately equally to the final distance, see Figure 4.7.

Removing the outliers from the whole dataset, using the ZMN method described above, the total number of unique players become approx 90.800 instead of approx. 94.000, removing $\approx$ 4% of the population. The five number summary for the whole dataset can be found in Table 4.3, notice the range of values compared to the Table 4.2 in Section 4.1.1. The multiple data batches that get feeded into the clustering algorithm had $\approx$ 5% player feature vectors removed on average, see Table 4.4 below.

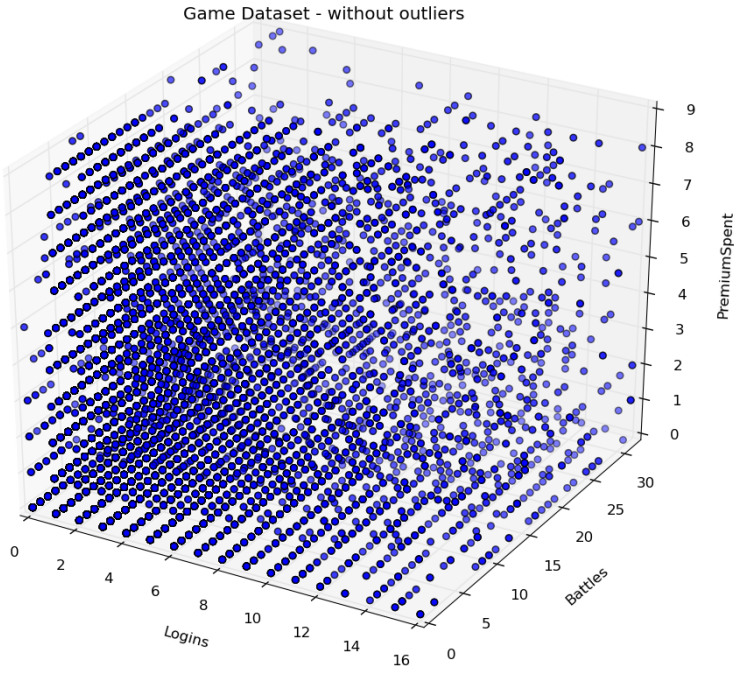


FIGURE 4.7: This figures shows the whole game metric dataset plotted in 3-dimensional data space, after applying the zscore transformation.

	Min	Q_1	*Median*	Q_3	*Max*
Logins	2	1	1	2	16
Battles	0	0	1	4	32
Premium spent	0	0	0	0	9

TABLE 4.3: This table shows the five number summary for the Logins, Battles and Premium spent events generated by each player from the whole dataset with outliers removed.

	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7	Data 8	Data 9	Data 10
Players	14054	13365	13403	13503	13441	13276	12839	13413	13473	13412
Outliers	658	741	661	673	748	698	685	724	669	680
% removed	4, 68%	5, 44%	4, 93%	4, 98%	5, 56%	5, 25%	5, 33%	5, 39%	4, 96%	5, 07%

TABLE 4.4: This table shows the player population, number of outliers found and the percentage of the population removed, for each data batch.

4.2 MapReduce K-means algorithm

A k-means algorithm was implemented in MapReduce (MR) that is able to incrementally process large-scale datasets in parallel when running on e.g. Amazon EMR. Three versions of MR k-means was implemented, two of them use different MR execution flows, but the final version builds on the second one and introduces a more efficient way of computing the nearest centroid in the Mapper phase. First version was the the naive one that introduces only a *Map* and *Reduce* phase, that assigns each point (feature

vector) to a nearest cluster in the Map phase and updates the centroids for the clusters in the Reduce phase. The programmer only needs to fill in two functions, the Mapper and Reducer.

The other two versions have a *Combiner* phase implemented that performs reduce work on the same computer node as the Mapper, by calculating intermediate sums of all the points for each cluster from a Mapper function. This minimizes the data needed to be transferred and shuffled by the MR framework to the Reducer, sending only an intermediate sum with each centroid from each mapper instead of list of all data points belonging to each cluster.

In the next subsections the three implementations and their pseudo codes are presented. Next will the *MR K-means Program* be explained, that controls the incremental data batch processing by executing n iterations of the MR K-means job. The most computation work in k-means is to find the nearest centroid for a given feature vector by calculating the distance between the two. Three different implementations were done for the nearest centroid function used by the Mapper.

4.2.1 MR K-means First Version - Map and Reduce

Implementing the Mapper and a Reducer function version is the most common implementation of k-means in MR, where the Mapper function calculates the nearest centroids to each data point and emits (sends) that pair to the Reducer function that sums up all data point values for each centroid, and outputs updated centroids. The Mapper function is called for each player feature vector found in the data batch, and iterates through all centroids that are stored in memory to calculate the distance to the nearest one. The nearest centroid and its data feature vector are sent from all the Map tasks(in parallel) to the MR framework that shuffles and sorts all feature vectors together which belongs to the same centroid. The pseudo-code for the Mapper function can be found in Algorithm 4.1 and for Reducer in Algorithm 4.2.

The Reducer function then gets called for each centroid with all of its feature vectors (assigned in the Mapper phase), iterates through all the feature vectors, sums up their values and divides with their count to get a new updated centroid and outputs that centroid, or the mean that represents the average feature vector for the cluster population.

Algorithm 4.1 K-means First Version: Mapper($key, value$)

Require: $key = document\_name$, $value = feature\_vector$
Ensure: $key = centroid\_id$, $value = feature\_vector$.

1: $current\_centroids \leftarrow C = \{c_1 ... c_k\}$ {loaded from memory}
2: $feature\_vector \leftarrow = value$
3: $nearestCentroid = getNearestCentroid(feature\_vector, current\_centroids)$
4: $EmitIntermediate(nearestCentroid, feature\_vector)$

Algorithm 4.2 K-means First Version: Reducer($key, values$)

Require: $key = centroid\_id$, $values = feature\_vector\_list$
Ensure: $key = centroid\_id$, $value = average\_feature\_vector$.

1: $feature\_vector\_list \leftarrow values$
2: $sum\_feature\_vector \leftarrow [0, 0, 0]$
3: **for all** $feature\_vector \in feature\_vector\_list$ **do**
4: $\quad sum\_feature\_vector \leftarrow sum\_feature\_vector + feature\_vector$
5: **end for**
6: $count\_feature\_vectors \leftarrow getCount(feature\_vector\_list)$
7: $average\_feature\_vector \leftarrow sum\_feature\_vector \div count\_feature\_vectors$
8: $Emit(key, average\_feature\_vector)$

Finding the nearest centroid to a point by computing the distances to all the k centroids is where the most of the work is done, and is be found in the Mapper phase. See Algorithm 4.3 to see the pseudo-code for the *getNearestCentroid()* function.

Algorithm 4.3 getNearestCentroid($point, centroids$)

Require: $point = 1 \times d$ *feature array*, $centroids = k \times d$ *array*).
Ensure: $nearestCentroid = nearest\_centroid\_id$.

1: $minDistance \leftarrow +\infty$
2: $nearestCentroid \leftarrow None$
3: **for all** *centroid* $c \in centroids$ **do**
4: $\quad distance = 0$
5: $\quad$ **for** *feature* i **to** d **do**
6: $\quad\quad distance \leftarrow distance + (point[i] - c[i])^2$ {sums up error for each feature}
7: $\quad$ **end for**
8: $\quad distance \leftarrow square\_root(distance)$
9: $\quad$ **if** $distance < minDistance$ **then**
10: $\quad\quad minDistance \leftarrow distance$
11: $\quad\quad nearestCentroid \leftarrow c$
12: $\quad$ **end if**
13: **end for**
14: **return** $nearestCentroid$

4.2.2 MR K-means Second Version - Map, Combiner and Reduce

By adding the Combiner phase in between the Map and Reduce phase, one can reduce a large amount of overhead in data transfer and computation in the MapReduce framework. The Combiner phase is executed on the same computer as the Map phase and is thought of as a reduce phase inside the Map phase. The Combiner function takes input directly from the Mapper function and calculates intermediate results that is emitted over the network instead of the Mapper emitting a key and value pair for each feature vector, which can be problematic for large datasets.

Algorithm 4.4 K-means Second Version: Combiner($key, values$)

Require: $key = centroid\_id$, $values = feature\_vector\_list$
Ensure: $key = centroid\_id$, $value = [sum\_feature\_vector, count\_feature\_vectors]$).

1: $feature\_vector\_list \leftarrow values$
2: $sum\_feature\_vector \leftarrow [0, 0, 0]$
3: **for all** $feature\_vector \in feature\_vector\_list$ **do**
4: $\quad sum\_feature\_vector \leftarrow sum\_feature\_vector + feature\_vector$
5: **end for**
6: $count\_feature\_vectors \leftarrow getCount(feature\_vector\_list)$
7: $EmitIntermediate(key, [sum\_feature\_vector, count\_feature\_vectors])$

In this version the same Mapper function as described above in Algorithm 4.1 is used but the Reducer needs to be changed. The Combiner function however, becomes very similar to the old Reducer in Algorithm 4.2. The only difference is that the Combiner doesn't compute the average feature vector or the mean, and emits a little more complex key and value pair, where $key = centroid\_id$ and $value = [sum\_feature\_vector, count\_feature\_vectors]$, see pseudo-code in Algorithm 4.4.

The new Reducer algorithm then sums up the intermediate sums and counts from the Combiners and calculates the average feature vector. The Reducer works with much smaller data as it receives a sum of values per centroid instead of all the feature vectors for each centroid, this is of great help when working with large data and has a much less communication overhead in the MR framework, the pseudo-code for the the Reducer is in Algorithm 4.5.

Algorithm 4.5 K-means Second Version: Reducer($key, values$)

Require: $key = centroid\_id$, $values = [sum\_feature\_vector, count\_feature\_vectors]$)
Ensure: $key = centroid\_id$, $value = average\_feature\_vector$.

```
1: sum_and_count_list ← values
2: sum_total_feature_vector ← [0, 0, 0]
3: count_total ← 0
4: for all sum_and_count ∈ sum_and_count_list do
5:     sum_vector ← getSumVector(sum_and_count)
6:     count ← getCount(sum_and_count)
7:     sum_total_feature_vector ← sum_total_feature_vector + sum_vector
8:     count_total ← count_total + count
9: end for
10: average_feature_vector ← sum_total_feature_vector ÷ count_total
11: Emit(key, average_feature_vector)
```

4.2.3 MR K-means Final Implementation

The final MR k-means implementation builds on the second version in Section 4.2.2 and differs only in computing the nearest centroid in the Mapper phase. calculating the nearest centroid is the most computation work in k-means.

Both the first and the second version of the MR k-means algorithm have a function that returns the nearest centroid to a data point by calling the function *getNearestCentroid()* Algorithm 4.3 that computes the Euclidean distance from a point to all the centroids and returns the centroid that has the minimum distance. This version of the function is the naive implementation to solve this problem, using time consuming for loops to calculate the distance to each of the centroids one at a time by iterating over all the features in the centroid and the feature vector, summing up their squared error.

A new version was implemented to optimize the work of calculating the nearest centroid and that version was used in the main experiments. Vectorized approaches from a Python scientific library called NumPy [1] were used. Optimizing calculation of the nearest centroid is a key to faster running times and if done right leads to better scalability.

NumPy introduces a new and powerful n-dimensional array data object, allowing more efficient computations to be done on array of objects, like array of n-dimensional feature vectors. Simple element-wise numerical operations are grouped together when applied

[1] http://www.numpy.org/

on these arrays, and they are implemented in the *C* programming language, giving them very high speed.

Vectorisation is applied and there is no need for a *for-loop* that iterates through all the centroids, instead its possible to find the nearest centroid using only few NumPy array operations. To find the nearest centroid to a point a new version of the *getNearestCentroid()* function was implemented for the MR k-means algorithm, the pseudo-code is found in Algorithm 4.6. The following steps are executed in the pseudo-code:

1. Each centroid vector is substracted from the point vector resulting in an $k \times d$ array that has the squared error for each feature for all the centroids.

2. Sums up each feature error for each centroid and returns an 1-dimensional array where each index represents the sum of squared error for each centroid.

3. The square root calculated for each centroid and returns the centroid id which has the smallest distance.

Algorithm 4.6 FINAL VERSION: GETNEARESTCENTROID($point, centroids$)

Require: $point = 1 \times d$ *feature array*, $centroids = k \times d$ *array*).
Ensure: $nearestCentroid = nearest\_centroid\_od$.

1: $errorPerFeatureArray \leftarrow (point - centroids)^2$
2: $errorPerCentroid \leftarrow SumArray(errorPerFeatureArray)$
3: $nearestCentroid \leftarrow MinArgArray(SqrtArray(errorPerCentroid))$
4: **return** $nearestCentroid$

The feature vector and the $k \times d$ centroids array are NumPy array objects. The functions *SumArray,MinArgArray* and *SqrtArray* are NumPy array functions and have been renamed for interpretability.

4.2.3.1 Alternative Nearest Centroid Computation - Not Used

There was one other approach that was implemented, trying to optimize the computation of the nearest centroid even further, but was not used in the final implementation. The idea is to have all the data (send to the Mapper) in memory then call do matrix and array operations on the data to get the similarity or dissimilarity matrix for all the data points to all the centroids, easily identifying the nearest centroid for all data points in

one operation. This was possible using a scientific Python library called SciPy [2] and the NumPy array object.

The problem with this solution that it needs to hold everything in memory and whereas the Map phase in MapReduce framework is designed to process one point at time and solve the computation problem for large data in parallel, e.g. splitting the data and run on hundreds or thousands of computers.

When this method was run locally (not using MapReduce implementation) it performed amazingly fast, e.g. running one iteration of k-means where it assigns all the points to a nearest centroid in a 880 MB (12.000.000 3-dimensional points) and calculates means of the clusters took only < 1 sek compared to ≈ 300 sek for the NumPy vectorisation method.

However there were problems when running the implementation in the Amazon EMR environment and the MR process failed many times or performed much worse compared to the vectorisation and the naive method. Tho managing to run when using more nodes (computers) in the EMR cluster set-up, but it performed much worse (8-22 time worse, when using 4, 6 and 8 nodes) than the naive and the NumPy vectorisation versions. This implementation was not used in the main experiments. It is included in an one experiment where the three nearest centroid computation methods have their running time compared using Amazon EMR.

The following steps were done to implement this version in MapReduce:

1. Mapper function gather one point at a time and stores in an array in memory. Emit nothing.
2. Mapper Final function was implemented that is called by the MR framework directly after the Mapper function finishes.
   (a) Calculate the distance matrix between all the data points and the centroids in memory.
   (b) Extract all the nearest centroids for all data points
   (c) Create a boolean filter for each centroid

[2] http://www.scipy.org/

(d) Gets all the data points by applying the boolean filter on the whole data point array.

(e) Calculates the intermediate sum for all points per centroid

(f) Emits a centroid id and the sum for its points. *(centroid_id, sum_points)*

3. Reducer will be called and sums up all intermediate sums from the Map phase as described before.

4.2.3.2 Memory Complexity

Comparing the memory complexity of the implementation using SciPy and using the NumPy vectorisation gives a good idea about the memory footprint the both versions need when doing these nearest centroid computations.

Storing the data point vectors in memory from the Map phase has a memory complexity $O(\frac{N \times d}{s})$ where s is the number of splits/Mappers, N is number of data points and d is the dimensionality. Generating the distance matrix takes $O(\frac{N \times k}{s})$ memory and store the nearest centroids for each point is $O(\frac{N}{s})$. Creating boolean arrays to filter out the points for each k centroids which then get sent as a key value pair to the reducers is $O(\frac{N \times k}{s})$. Putting this all together, needs at least $O(\frac{N \times d}{s} + \frac{N \times k}{s})$ memory, and compared to the first vectorisation method with NumPy it would need about $O(d)$ for the incoming data point and $O(k)$ for the distances to each of the k centroids with total memory complexity of $O(d + k)$.

4.2.4 MapReduce K-means Program

This program controls the iteration process of executing the MR k-means job algorithm in each iteration. It executes on a data batch basis, when a data batch is ready to be processed, then this program is called and it returns the current intermediate means found. When the next data batch is ready the program is executed and it expects to receive the previous found intermediate means as initial means for the current incoming data batch. The program expects following inputs:

- *input_file*: Path to data batch file (can also be an Amazon S3 URI). The file is read by the MapReduce framework, split and distributes the data batch to the Mappers.
- *initial_means*: Path to a means file. If running the program for the first time, then these means must be selected from the data batch. Otherwise this is the intermediate means output from the previous data batch processing.
- *k_means*: The number of centroids to be found in the data batch.
- *max_iterations*: The number of iterations to run k-means on the data batch.
- *threshold_delta*: A value that defines the minimum centroids difference/movement found between iterations. (e.g. 0.01)

When setting the maximum iterations for a data batch too high, there is a risk of k-means adjusting the intermediate means to close to the distribution of the current data batch and can be costly when they are used as input to the next data batch that has a slightly different distribution. See Algorithm 4.7 for the pseudo-code of the program.

Algorithm 4.7 MapReduce Kmeans Program()

Require: $input\_file$, $initial\_means$, k_means, $max\_iterations$, $threshold\_delta$
Ensure: $means$

```
1:  intermediate_means ← initial_means
2:  for i to max_iterations do
3:      MRjob_parameters ← [input_file, intermediate_means, k_means]
4:      output ← runMRJob(MRjob_parameters)
5:      means ← getMeans(output)
6:      means_difference ← dist(intermediate_means, means)
7:      if means_difference < threshold_delta then
8:          return means
9:      else
10:         intermediate_means ← means
11:     end if
12: end for
13: return intermediate_means
```

The main steps in the execution flow when running MR k-means program incrementally, see Figure 4.8, are described as following:

- 1. Input: Means or centroids found from previous data batch processing iteration and the current batch of data to be processed.
- 2. MR k-means: Run n-iterations of MR k-means job. In each iteration; The Map/Combine phase *assigns* data points to nearest means by sending the nearest centroid for each point, then the Reduce phase *updates* the means of the assigned points. If more than 1-iteration is performed, the intermediate means from the previous iteration is used as input to the MR k-means job.
- 3. Output: After n-iterations the program outputs the means found for the current data batch.
- 4. Process: If there is multiple data batches to be processed then the program is run again with means from step 3. as input and the next data batch file.

4.3 Development Environment and Tools

The MapReduce k-means algorithm was implemented using a MR development framework called *mrjob* [3]. Mrjob is a open-source Python framework actively maintained by Yelp [4], that allows MapReduce jobs to be written in Python 2.5+ and executed on several platforms. Using mrjob allows for rapid implementation of MapReduce jobs by running them locally for development purposes and easily run them on your own Hadoop cluster or even easier using the Amazon Elastic MapReduce (Amazon EMR)[5]. Amazon EMR is a web service that allows developers to buy time on a Hadoop cluster to process large-scale data easily and cost-effectively. Amazon EMR web service was used in the scalability and the nearest centroid experiments.

The Python programming language was chosen for this study because GameAnalytics (GA) also uses Python and mrjob to implement their MapReduce jobs. Allowing GA easily to use and build further on the implementation from this study. In this thesis, the NumPy/SciPy Python library that enables vectorisation approaches and array/-matrix data structures for easier and more efficient data vector manipulations. The

[3] http://pythonhosted.org/mrjob/
[4] http://opensource.yelp.com/
[5] http://aws.amazon.com/elasticmapreduce/

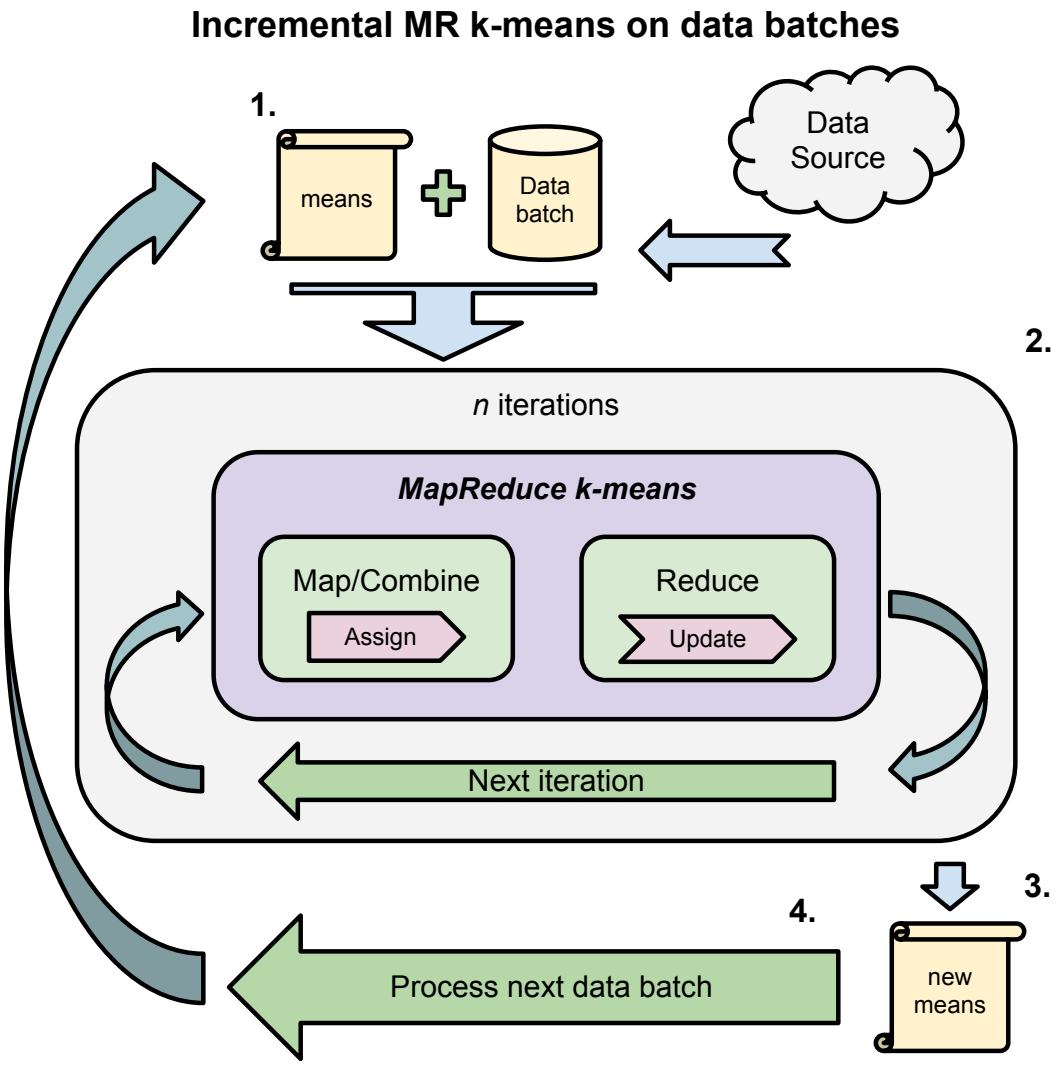


FIGURE 4.8: The incremental MR k-means execution flow.

implementation of algorithms and running experiments were executed on a Linux operating system, recommended by GA instead of using a Windows OS, because of Python libraries compatibility and mrjob.

4.4 Experiments Set-up

Several experiments were conducted to evaluate the MapReduce k-means algorithm, the experiments were:

1. **Cluster Quality - Real Dataset**: Test the cluster quality and error stability when clustering incrementally multiple data batches of real game data.
2. **Cluster Quality - Generated dataset**: Same as the first experiment, but using a generated multiple data batches with three normal distributions were the means are moving and changing slowly between batches.
3. **General Player Behaviour**: Find and interpret the general player behaviour in the real game data, after incrementally clustering multiple data batches with MR K-means.
4. **Scalability**: Measure the *horizontal* (scale-out), *vertical* (scale-up) and scaling both computing power and data size. Clustering a single set of generated dataset of different sizes and on different combinations of computers and processors. All measures are run using the Amazon EMR web service.
5. **MR K-means Combiner vs. non-Combiner**: Compare the running time when the Combiner function is not implemented in the MapReduce execution flow.
6. **Nearest Centroid Computation Methods**: Compare three different nearest centroid computation methods using Amazon EMR.

4.4.1 Experiment: Cluster Quality - Real Dataset

For this experiment the cluster quality or Sum of Squared Error (SSE) between multiple data batches was measured and how stable the error was. As described in Section 4.1.2 the user telemetry dataset needed to be preprocessed and split into ten data batches, where the data was timely ordered after the *event* timestamp. Ten parts sounded a reasonable number and was therefor used for this experiment, leaving ≈ 510.000 events for each part. For each part events were aggregated or counted to construct the game metric datasets used as input to the algorithm, each containing $\approx$ 13.500 records of unique players, see Section 4.1.1 for the game metric feature selection.

Having these ten timely ordered datasets we are simulating the environment were new chunks of game data arrive each day from a game, like in GameAnalytics. In Figure 4.9 we can see an example how four of the ten data batches look like in 2-d, after running the normal iterative k-means implementation in the Python scientific library SciPy.

The MR k-means algorithm in Section 4.2.2 was executed locally instead of using Amazon EMR. Using the Python mrjob MR development framework the MapReduce environment and its execution flow is simulated locally instead running it on e.g. Hadoop instances in Amazon EMR, this was done to save time and should not affect the results.

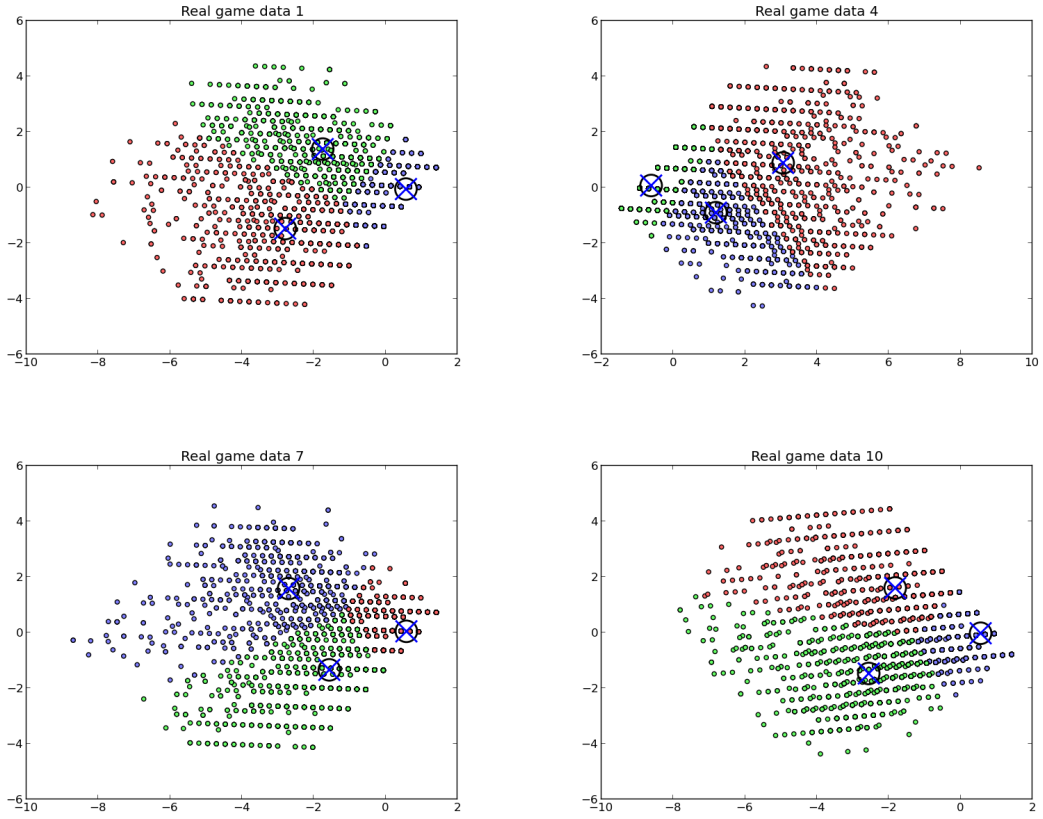


FIGURE 4.9: In this figure we can see four of the ten data batches that were used in the experiment when measuring the cluster quality for multiple real game datasets. The data batches are 3-dimensional and were projected to 2-dimensional for better visualization, using Principal component analysis.

Three different methods were compared to measure the cluster quality for each data batch over multiple batches. These methods test different number of iterations that we run the MR k-means algorithm on each data batch, the methods are following:

- 1-iteration

- 2-iterations
- 5-iterations

Ten experiments were performed each with different random initial centroids selected from the first data batch. For all the methods we start with the same initial centroids and parameters; number of centroids $k = 3$ and the minimum change in centroids, $threshold\_delta = 0.01$, if below threshold the iteration loop exits before finishing the n iterations. Each experiment executes the following steps:

1. Continue until no more data batches.
2. Select the next data batch.
3. Select intermediate centroids *(If the first batch, then use the initial centroids).*
4. Execute the MapReduce k-means program with a n-iteration method of choice.
5. Calculate the SSE.
6. Store the intermediate centroids received after n-iterations.

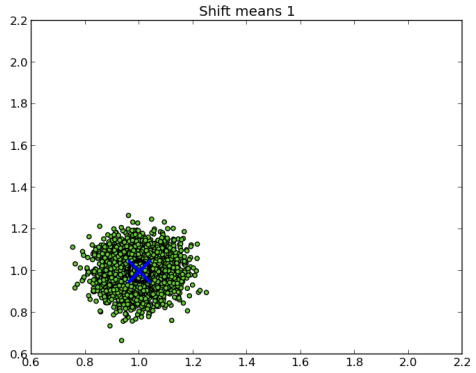
The MapReduce k-means program (see Algorithm 4.7) controls the iterations of the MR k-means algorithm (see Section 4.2.2).

4.4.2 Experiment: Cluster Quality - Generated Dataset

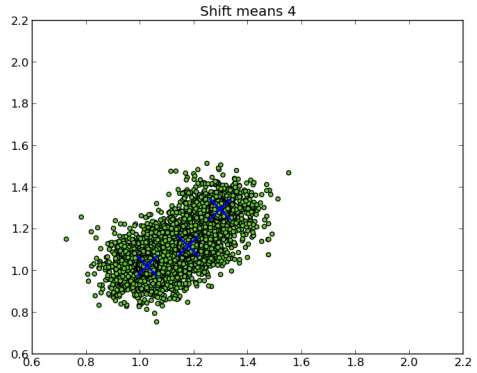
Same measurements, setups and execution flow were performed for this experiment like the experiment in Section 4.4.1, except generated data batches were used instead of the real game data set. The generated dataset was also split up into ten data batches, each containing 3.000 2-dimensional records.

The idea with this experiment is that the fictive data was generated with three normal distributions where they move between data batches. In the first data batch all three means are very close together. In the next data batches the means move little bit further away from each other and also the population in two of the clusters changes over time, where the furthest one (up-right) loses data points while the lower to the left gains those points, this was tried to simulate that less extremes are found in reality than the

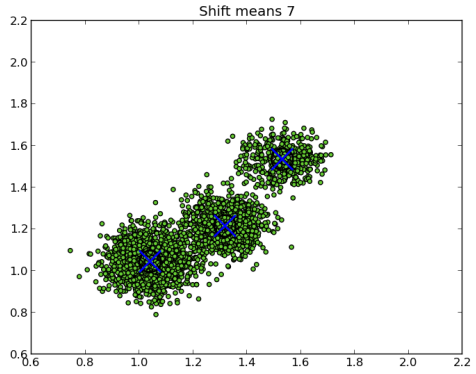
majority of the population. We can see in Figure 4.10 how the means move between data batches.



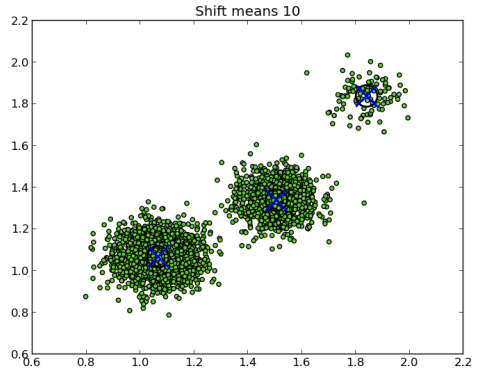
(a) Three means start close together



(b) The means are moving a little bit



(c) Three distributions can now be spotted



(d) One mean has moved away from the crowd and with fewer points

FIGURE 4.10: In this figure we can see four of the ten datasets that were used in the experiment when measuring the cluster quality for multiple generated datasets.

As mentioned the experiment was executed with same setup, parameters and the same methods were compared as described in the other cluster quality experiment for the real data in Section 4.4.1.

4.4.3 Experiment: General Player Behaviour

In this experiment three general player behaviour groups were extracted and interpreted from the the real game dataset in Section 4.1, contributed by GameAnalytics.

The results from the cluster quality experiment in Section 4.4.1 were used and analyzed for this experiment. The n-iteration method that gave the most stable SSE error results

was selected, where multiple data batches were clustered incrementally. The final means that had the lowest SSE after the last data batch was used to represent the general player behaviour for the data batches.

The means uses were z-score values (Standardize values, see preprocessing in Section 4.1.2) and were converted to the original range of values, by transforming z-score formula to find the original value as follows

$$x = (z * \sigma) + \mu$$

where x is the original value in the original numeric range and z is the z-score value. The standard deviation σ and average μ were calculated from the original range for that data batch.

4.4.4 Experiment: Scalability

High scalability of the MR k-means algorithm is very important when the datasets increases in sizes. The MapReduce framework offers high scalability out of the box in its execution flow, favoring scale-out instead of scale-up with the idea of distributing a large-scale work to a large number of commodity low-end servers instead of a few expensive high-end servers. Multiple computers work on each piece of the data in parallel and all the intermediate results in a fault tolerance are combined in the end. An algorithm must also be efficient and well written with parallel processing in mind to take a full advantage of the MapReduce framework.

All experiments were run using The Amazon Elastic MapReduce (Amazon EMR) Web Service [6]. Offering vast possibilities to easily configure and run various Hadoop clusters instances on the Amazon Elastic Compute Cloud (Amazon EC2) on-demand.

All datasets were generated with three normal distributions in 3-dimension. The data points were shuffled (using NumPy shuffle) such that they represent a random order internally, avoiding bias when the MapReduce framework is splitting up the data to be distributed. All datasets were uploaded to the Amazon Simple Storage (Amazon S3) providing fast and safe data transfer to and from the Amazon EC2 Hadoop instances.

[6] http://aws.amazon.com/elasticmapreduce/

The MR k-means algorithm (in Section 4.2.2) was measured using the 1-iteration method, that is the algorithm runs only one iteration on the dataset. The running time of the algorithm was taken from the MapReduce system log file where it states when the job starts running and completes, excluding e.g. the extra overhead when starting the MR job directly from the MR k-means program, see Section 4.2.4.

The scalability of the MR k-means algorithm was measured by conducting the following experiments:

- Horizontal scaling (Scale-out): Number of computer nodes are doubled, same dataset size.
  - Dataset: 220 MB.
  - EC2 Type: 5 Compute Unit (2 cores x 2.5 units), High-CPU Medium.
  - Nodes setup: 2, 4, 6 and 8.
- Vertical scaling (Scale-up): The computing power is doubled, same dataset size.
  - Dataset: 220 MB.
  - EC2 Type: 1 Compute Unit (1 core x 1 unit), M1 Small.
  - EC2 Type: 2 Compute Unit (1 core x 2 units), M1 Medium.
  - EC2 Type: 4 Compute Unit (2 cores x 2 units), M1 Large.
  - EC2 Type: 8 Compute Unit (4 cores x 2 units), M1 Extra Large.
  - Nodes setup: 2
- Scaling: The dataset is doubled in parallel when doubling the number of processor nodes.
  - Dataset: 220 MB and 2 nodes.
  - DataSet: 440 MB and 4 nodes.
  - Dataset: 880 MB and 8 nodes.
  - Dataset: 1.760 MB and 16 nodes.
  - EC2 Type: 5 Compute Unit (2 cores x 2.5 units), High-CPU Medium.

4.4.5 Experiment: MR K-means Combiner vs. non-Combiner

Compared the running time for the MR k-means algorithm with the Combiner function removed from the MR execution flow, see Section 4.2.2 for the algorithm details.

The Amazon EMR web service was used for measuring the running time for the two algorithms. The cluster setup was the same for both algorithms when processing different sizes of datasets. The datasets are the same as explained in the experiment in Section 4.4.4, also the running time was taken from the MapReduce system log inside Amazon EMR.

The experiment set-up:

- Dataset: 220 MB, 440 MB, 880 MB and 1.760 MB.
- EC2 Type: 20 Compute Unit (8 cores x 2.5 units). High-CPU Extra Large.
- Nodes setup: 4

4.4.6 Experiment: Nearest Centroid Computation Methods

The three different methods of computing the nearest centroid, that is finding the centroid that has the minimum Euclidean distance between a data point in a dataset, were measured and compared. The most computation work in the MR k-means algorithms is finding the nearest centroid for each point that is being processed in the Mapper function. The implementation of the methods are explained in Section 4.2.3 and 4.2.3.1. They are:

- Naive version: For loops, iterating over all features for all k centroids calculating the squared error for each data point.
- Vectorisation - NumPy: Using a special NumPy n-dimensional array object, allowing efficient operations without use of for loops.
- Distance matrix - SciPy: Loading all the data points into memory and find the distance matrix to find nearest centroids for all data points in a super fast way but with high memory cost.

Two experiments were conducted on Amazon EMR:

- Dataset: 220MB.
- EC2 Type: 5 Compute Unit (2 cores x 2.5 units), High-CPU Medium.
- Nodes setup: 4, 6, 8, 10

The datasets are the same as explained in the experiment in Section 4.4.4, also the running time was taken from the MapReduce system log inside Amazon EMR.

4.4.7 About the experiments

Here are some general notes when the experiments were conducted:

- It takes time to fire up a MR job on a new on-demand EC2 instance on Amazon EMR. The process of starting the instance, bootstrapping can take up to 5 minutes. To safe time a *Job Flow* was created and used, to start an EC2 instance with out a MR job. After that it is possible to run jobs quickly by assigning them to a available job flow. (Useful when doing e.g. n-iterations of the MR k-means job)
- The default EC2 instance type for the Master Controller node for all the experiments when running on Amazon EMR was: Compute Unit 1 (1 core x 1 unit), M1 Small.
- Config file for mrjob: Needed to set the Hadoop version to 1.0.3 for all instances to be able have the newest batches/updates, such as NumPy/SciPy scientific libraries.
- The parameters for number of centroids $k = 3$ and the $threshold\_delta = 0.01$ was kept from the start of the implementation and to the end of the experimentation process. The idea was to focus on few groups than too many and stop a k-means iteration process if the means moved less than 1% between iterations. Tuning these parameters were not under a scope for this thesis.
- The initial k centroids for the MR k-means algorithm was always picked randomly from the dataset under test. When doing comparisons then the initial centroids were stored beforehand and tests under question would use the same centroids.

Chapter 5

Results

In this chapter the results from the conducted experiments from Section 4.4 are presented.

5.1 Cluster Quality - Real Dataset

Here are the result after incrementally cluster the ten multiple data batches of real game data and measuring the SSE error after performing n-iterations on each data batch. The SSE error gives an idea of the cluster quality at each data batch when using the k-means centroids results from previous data batch as an input.

Three different variations of n were compared to see if there is any difference in doing a few iterations versus many iterations over each data batch. The main idea is to have a method that has a stable SSE error over multiple batches of data and have the desirable feature of having a downward trend for the SSE.

Ten test runs were performed with different initial centroids from the first data batch trying to have the centroids to start at different points in space, creating different clusters and centroids as output from the first batch which can affect the consecutively batches.

As we can see from Figure 5.1 all the methods have a downward trend but are increasing the error with different amounts in batch $7 - 10$, tho the *Iteration 1* method is showing a more stable change towards the final batch.

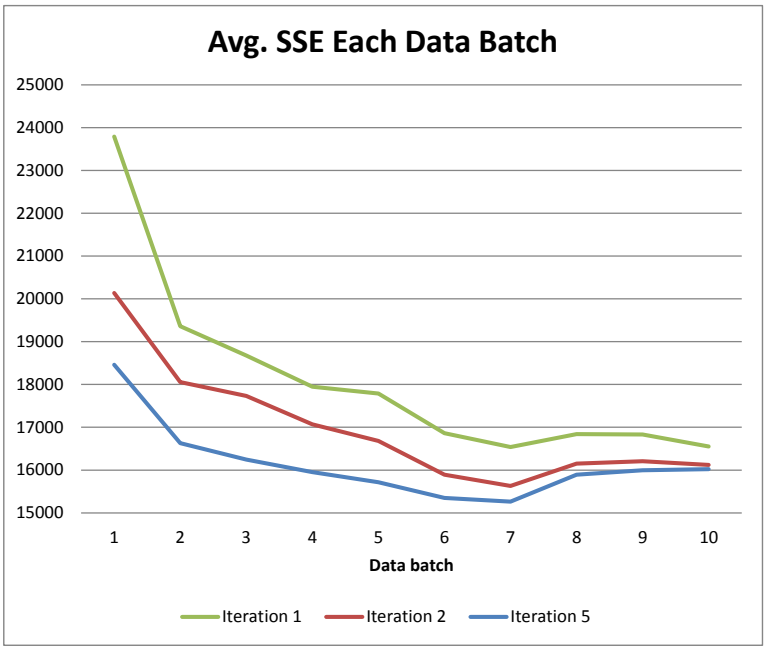


FIGURE 5.1: This figure compares the average SSE error after clustering each data batch, using the three different n-iterations methods.

After incrementally clustering the ten data batches with the different n-iterations methods, they all represent very similar clusters in the last batch, see Table 5.1. The *Iteration 2* and *Iteration 5* has only about 0.6% difference even tho the *Iteration 5* is constantly showing lower SSE error in the first 7 batches but then takes a steeper SSE increase up to the last batch. In this case the *Iteration 5* method was too aggressive by trying to fit the centroids from the previous batch to the current batch by doing 5-iterations.

	Iteration 2	*Iteration 5*
Iteration 1	2.60%	3.19%
Iteration 2		0.60%

TABLE 5.1: This table shows the average SSE error difference on the last data batch, between the three methods.

In Figure 5.2 we can see on average how much was a single increase between one set of data batches for all test runs. The average single increase for the *Iteration 1* method in each test is very different than the other methods. The reason is that are different initial centroids for each test and the *Iteration 1* method only performs one iteration each time when clustering a data batch, doing so the method takes only one step towards the current data batch, slowly changing the means. If a data batch introduces a much different distribution this method will give worse results than the others because they

are taking more steps towards the new data, but risking to deviate too much from the general population seen so far.

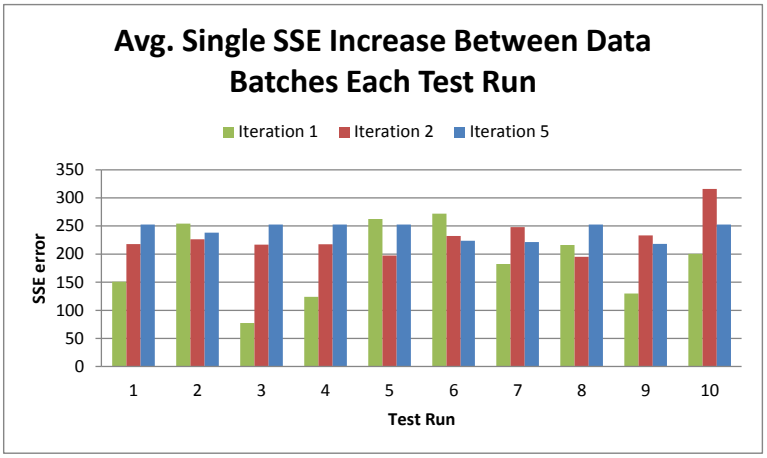


FIGURE 5.2: Here we can see on average the single SSE error increase that was measured, when SSE increases from one batch to another, for all the test runs.

When looking at the Figure 5.3 we can see the single SSE error increase on average over all the ten test runs. The *Iteration 1* method has considerable lower single SSE error increase compared to the *Iteration 2* and*Iteration 5* methods. The latter two methods being very similar to each other, both moving the centroids more aggressively away from already seen centroids.

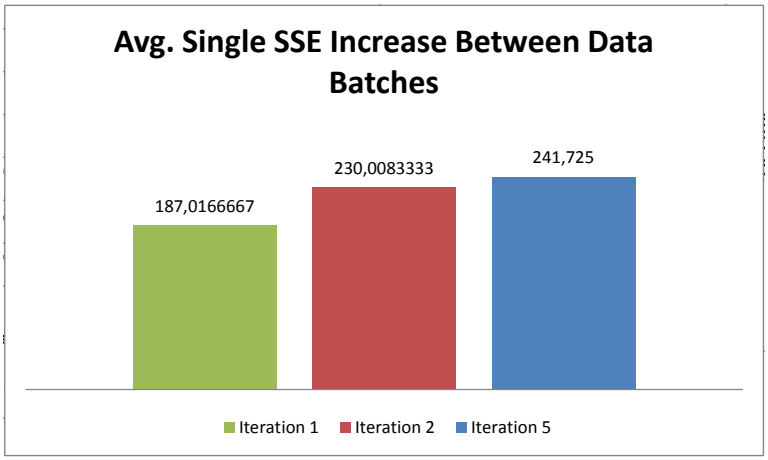


FIGURE 5.3: Here we can see the average SSE error increase between two data batches, for the all the test runs.

The results show that the *Iteration 1* method has the lowest average single SSE error increase between a set of data batches. We now look at the aggregated SSE increase for each of the test runs, see Figure 5.4.

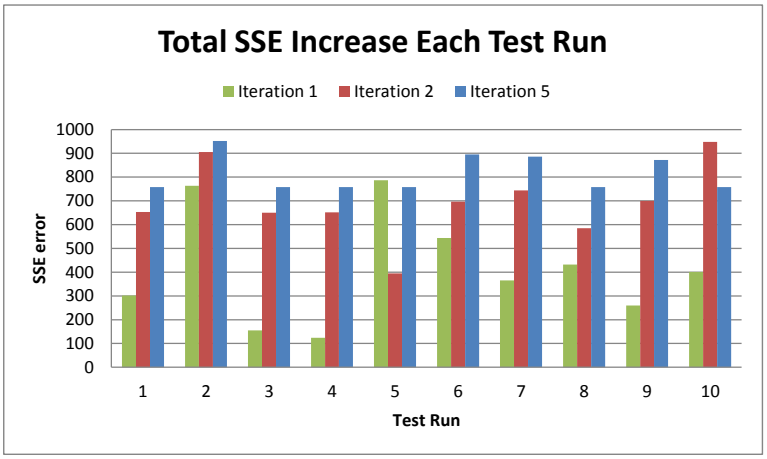


FIGURE 5.4: This figure shows the aggregated SSE error between two data batches for each test run.

The *Iteration 1* method shows the lowest aggregated SSE error increase in all test runs, except in test run 5. The random initial centroids from the first batch poorly represents the main distribution in the data and by performing 1-iteration takes a longer time to adjust the centroids towards the mean of population. In Figure 5.5 is the average aggregate SSE error shown, over all the ten runs peformed.

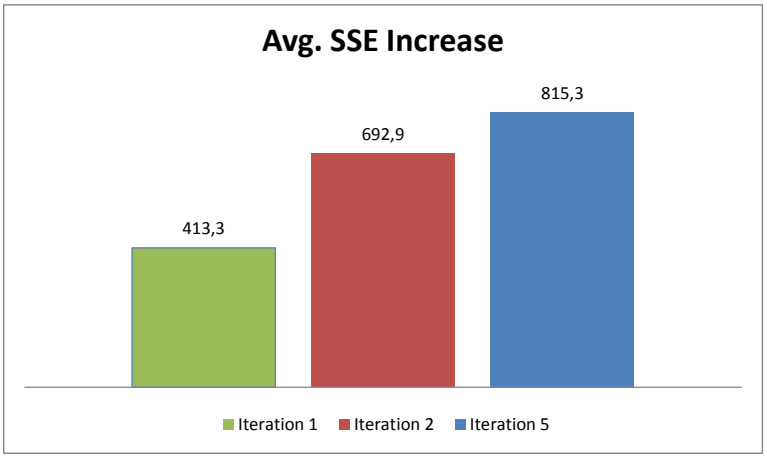


FIGURE 5.5: This figure shows the average aggregated SSE error between two data batches, for all test runs.

5.2 Cluster Quality - Generated Dataset

In the second cluster quality experiment a generated dataset is used. This dataset is generated with three normal distributions that changes and move from each other in the course of ten data batches. The idea is to measure the SSE error when facing a rapid changing data, in that sense that the distributions are moving between chunks of data.

Finding a method that can represent a stable SSE error and a downward trend is of high value, as explained in the Section 5.1. The Figure 5.6 shows the SSE error that was measured on average for all the ten test runs when clustering the ten data batches.

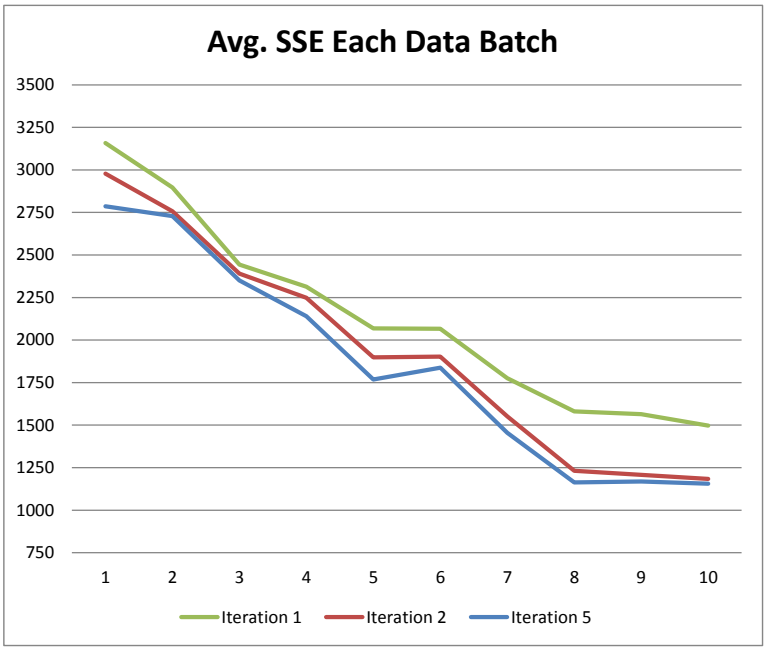


FIGURE 5.6: This figure compares the average SSE error after clustering each generated data batch, using the three different n-iterations methods.

We can see that both the *Iteration 1* and *Iterantion 2* methods show a downward trend for the SSE error but the *Iteration 5* method is taking a relatively big step from data batch 5 to 6. Methods *Iteration 2* and *Iteration 3* show the lowest SSE error for the last data batch, in Table 5.2, with only 2.36% difference, and there is large gap of 22.78% between *Iteration 1* and *Iteration 5*.

This large gap between *Iteration 1* and the other methods is because in this dataset the three distributions are changing relatively fast, moving it in a determined direction in each data batch. Making it hard to keep up with the changes with 1-iteration while the other methods are moving closer to each next data batch.

	Iteration 2	*Iteration 5*
Iteration 1	20.91%	22.78%
Iteration 2		2.36%

TABLE 5.2: This table shows the average SSE error difference on the last generated data batch, between the three methods.

In Figure 5.7 we can see the measured SSE error when it increases between data batches, e.g. where a SSE error for a batch is higher than the previous data batch. The *Iteration 2* method is now showing the lowest error on average, and in 4 out of 10 tests it almost doesn't increase the SSE. Method *Iteration 5* is however very interesting in test 3 where it much lower than the other methods.

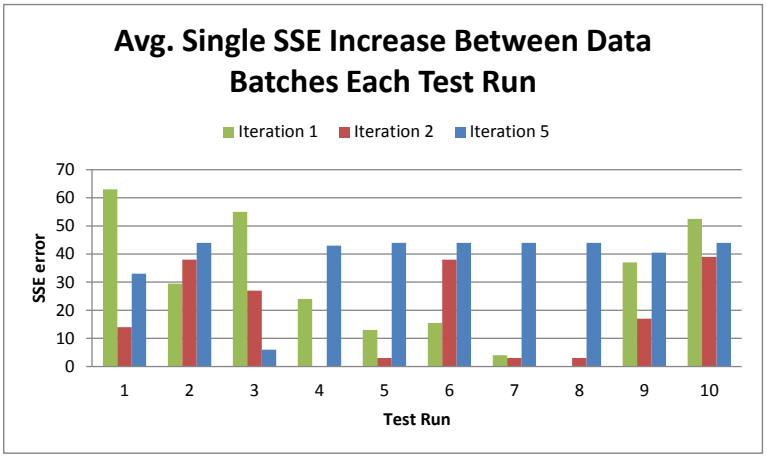


FIGURE 5.7: Here we can see on average the single SSE error increase that was measured, when SSE increases from one batch to another.

Looking into the data it shows that the *Iteration 5* method manages to report much lower SSE error than the other methods when processing data batch number 6, using the centroids found in data batch number 5. And in general all the methods had a SSE increase when processing batch 6, see Figure 5.8 which data batches gave problems over all the tests. Batch 6, 9 and 10 gave high SSE increase on average, but the batch 6 had the far most error for all the methods.

When looking at the data population for data batch 5 and 6 we can see that there is a clear change in the data, going from batch 5 to 6, see Figure 5.9. The figure on to the right shows clearly that one of the clusters have disconnected from the rest, and this seems to give problems for all methods. *Iteration 5* had the most troubles with batch 6

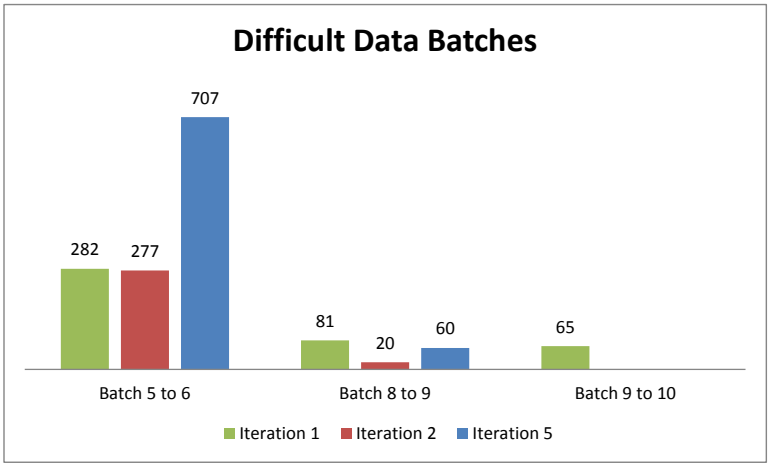


FIGURE 5.8: This figure shows the data batches where all methods increased the SSE error going from one data batch to the next.

in general, most likely by moving its centroids to aggressively through many iterations and end up in a local optimum.

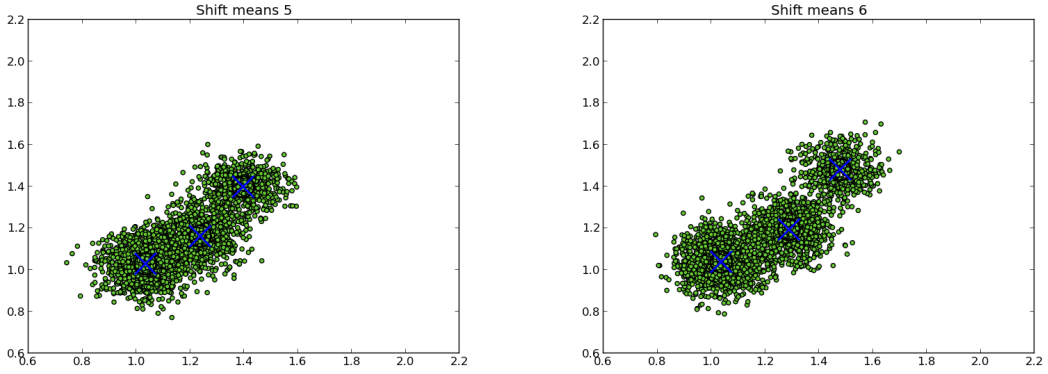


FIGURE 5.9: These figures shows how the data distribution changes from data batch 5 to 6. In the right figure we can see a cluster that has moved from the other two clusters.

For the *Iteration 5* method from test 3 in Figure 5.7, it is likely that the random initial centroids for the first batch, leads the centroids very close to the highest population such that it didn't fell in to a trap when processing batch number 6, thus measuring low SSE for that method. The same happens in general for the other methods since they move much slower and are closer to the main population.

Looking at the measured SSE error that happened between all sets of data batches, we can see in Figure 5.10 that *Iteration 2* is reporting over all tests very low SSE increase

but is high in tests 2, 6 and 10, which were related to the data batch number 6, discussed above.

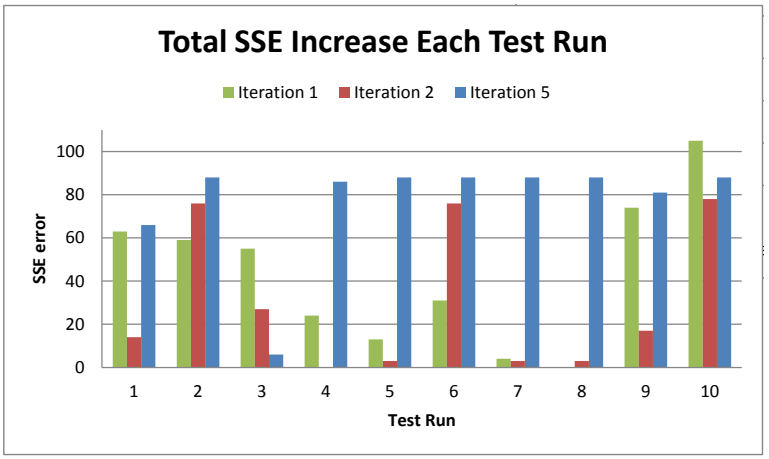


FIGURE 5.10: This figure shows the aggregated SSE error between two data batches for each test run.

When aggregating all the SSE error between all the sets of data batches over all runs we get *Iteration 2* with the lowest average, with *Iteration 1* not far away.

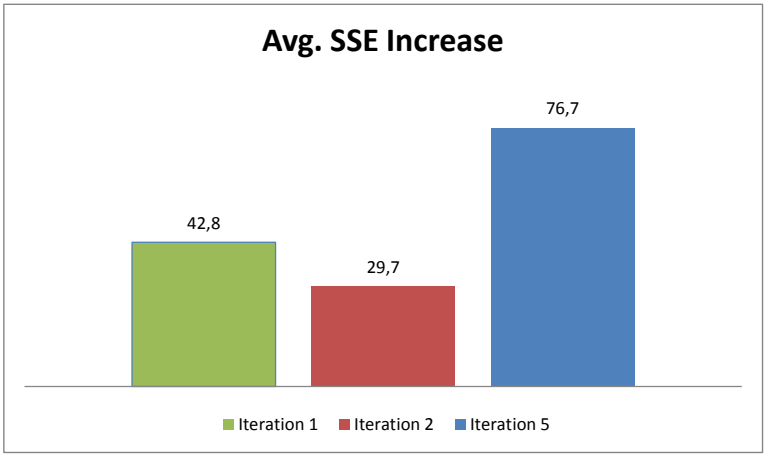


FIGURE 5.11: This figure shows the average aggregated SSE error between two data batches, for all test runs.

5.3 General Player Behaviour

The results from real game data cluster quality experiment in Section 5.1 were used and interpreted to find the general player behaviour. The centroids from the last data batch when using the *Iteration 1* method was used as basis for this experiment, from the test run that had the the lowest SSE.

The three centroids from the final data batch, found by the MR k-means algorithm, describes the average behaviour of the three clusters found, after clustering incrementally the ten data batches. See Table 5.3 for the centroids and their z-score values, and Figure 5.12 for the visualization of the clusters found.

	Logins	*Battles*	*Premium spent*
Centroid 1	2.25	1.35	0.06
Centroid 2	−0.29	−0.33	−0.31
Centroid 3	0.24	1.46	2.62

TABLE 5.3: This tables shows the how the final centroids in *z-score* values looked like after clustering the final game data batch file, using the *Iteration 1* method.

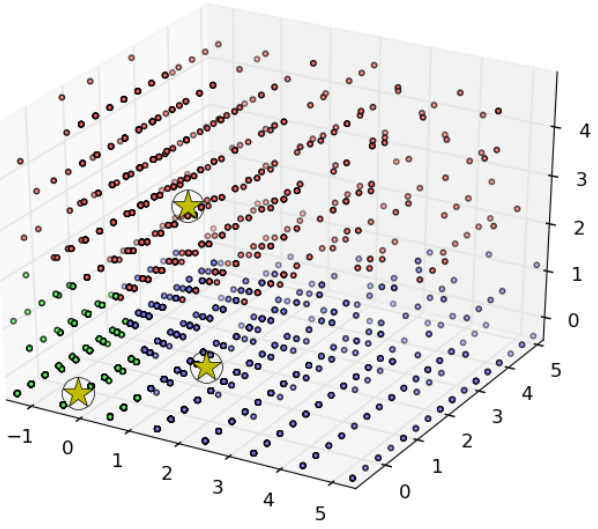


FIGURE 5.12: This figures shows how the clusters looked like for the final data batch, the centroids are marked with stars.

These standardized values were transformed to their original values as explained in the experiment set-up in Section 4.4.3. For the original values see Table 5.4. The centroids represent the general player behaviour based on the behavioural features that were used, and can be used to describe behavioural profiles that is found in the data. See Table 5.5

for the player profiles that were extracted and interpreted from the centroids. The profile titles and their descriptions were made from a guess and intuition.

	Logins	*Battles*	*Premium spent*
Centroid 1	3.80	2.86	1.51
Centroid 2	1.14	1.10	1.12
Centroid 3	1.70	2.97	4.19

TABLE 5.4: This tables shows the centroids in the original range of values for each game metric, used for interpretability.

Profile title	*% of P*	*Description*
Active Joe	9.39%	Explores areas and fights battles
Lazy	81.12%	Jumps in and out of the game
Golden Warrior	9.49%	Fights battles and spends in-game money

TABLE 5.5: This table shows the player profiles and their size in the player population. Extracted from the centroids that describe the k average player feature vectors after incrementally clustering multiple batches of game data.

5.4 Scalability

The results from the scalability experiment shows that the MR k-means algorithm is scalable algorithm that is fully capable to be run on Amazon EMR web service, launching on-demand Amazon Elastic Hadoop clusters. In Figure 5.13 we can see the vertical scaling, where the computing power is doubled but processing the same size of data.

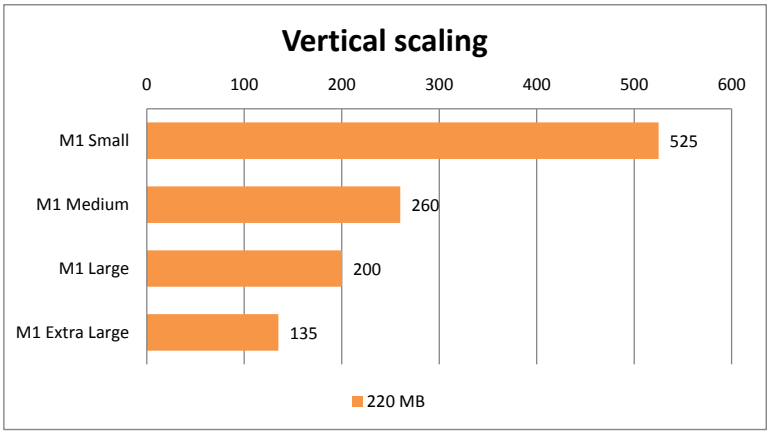


FIGURE 5.13: This figure shows the running time (in sec) when doing vertical scaling for the MR k-means algorithm. Doubling the computing power on same dataset size.

The horizontal scaling results are in Figure 5.14, and show the running time when doubling the number of computers or nodes when clustering the same size of data.

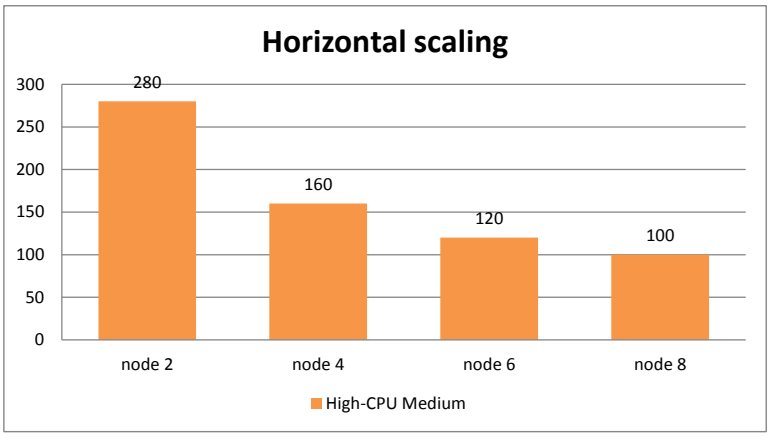


FIGURE 5.14: This figure shows the running time (in sec) when doing a horizontal scaling for the MR k-means algorithm. The number of computer nodes are doubled each time on the same size of data.

The MR k-means scales both horizontally and vertically, showing the biggest performance jumps when doubling in the beginning of the tests. In Table 5.6 we can see how the MR k-means scales when doubling both the number of computer nodes and dataset sizes, resulting in nearly the same execution time for all tests.

Size	*# nodes*	*time in sec*
220 MB	2	280
440 MB	4	280
880 MB	8	285
1760 MB	16	285

TABLE 5.6: This table shows the execution time for the MR k-means algorithm when doubling the computer nodes and the dataset size.

5.5 MR K-means Combiner vs. non-Combiner

Here we look at the running time results the final MR k-means algorithm was compared with it self but with the Combiner function removed from the MapReduce execution flow. The results show that the Combiner is an essential feature when there is a possibility to reduce problems in the Mapper phase, thus minimizing the data traffic and allowing more efficiency in the MapReduce framework, see Figure

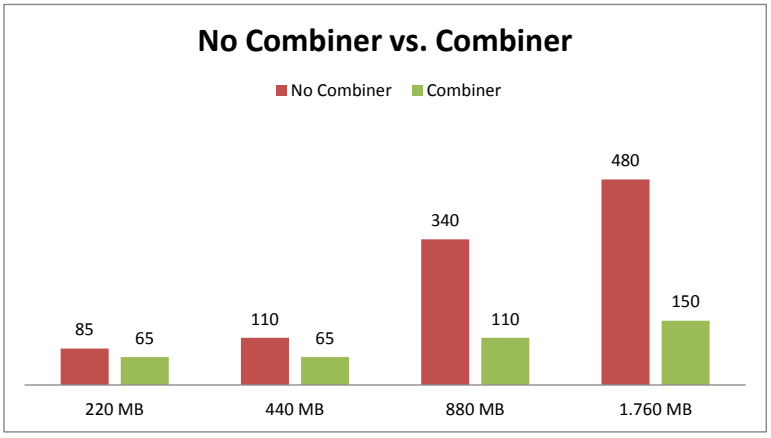


FIGURE 5.15: This figure compares the MR k-means algorithm with and without the Combiner function in MR implemented. We can see that having a Combiner can quickly become very efficient versus the non-Combiner version.

5.6 Nearest Centroid Computation Methods

The execution time for the three nearest centroid computation methods were compared. The Mapper phase in MapReduce is responsible to find the nearest centroid for each data point, and this computation is the most expensive work when running k-means.

The running time for the naive and the vectorisation versions were compared, leaving the distance matrix implementation out of the picture, for now. See Figure 5.16 for the comparison of the two. On the figure we can see that the vectorisation version is performing much faster but it converges when running on 8 computer nodes, the MapReduce framework overhead takes over and to make the algorithm run faster we can scale-up by upgrading the computing power.

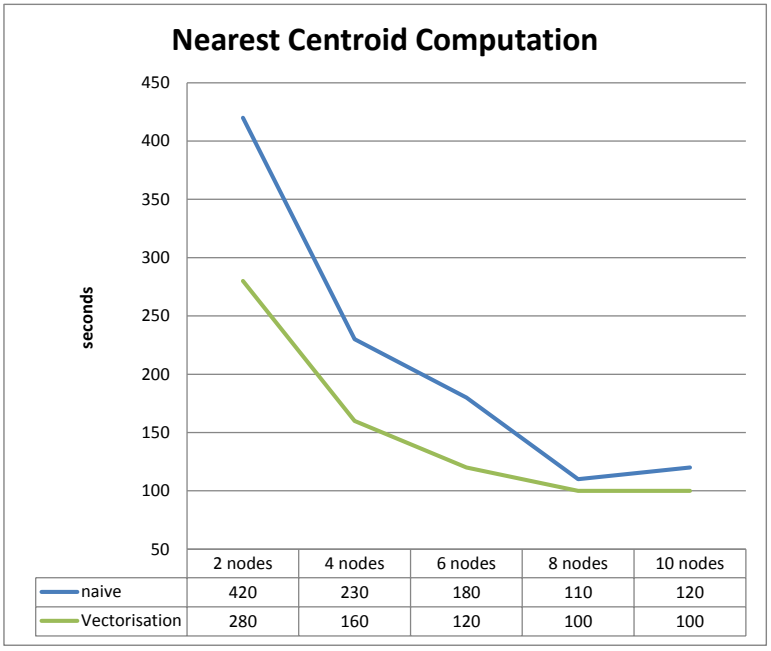


FIGURE 5.16: This figure compares the execution time for the naive and vectorisation methods when computing the nearest centroid.

Running the distance matrix version to compute the nearest centroid doesn't work so well on Amazon EMR, compared to the other methods, see Figure 5.17 for the running time for the distance matrix version. Running with 2 nodes resulted in error in the MR execution process and other execution times are much higher, e.g. running with 8 nodes in this setup the distance matrix is ≈ 10 times slower.

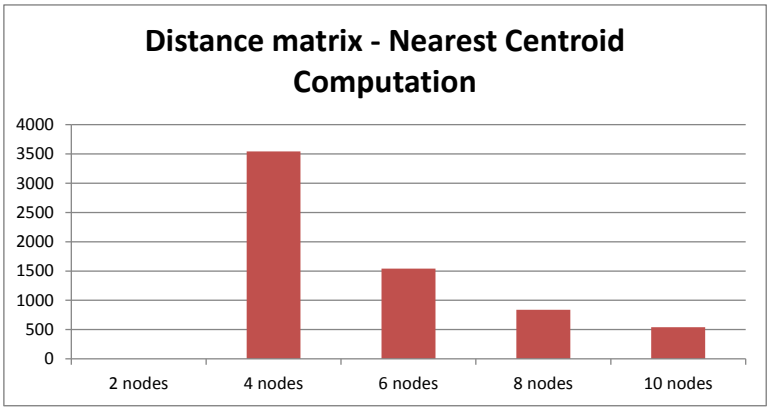


FIGURE 5.17: This figures shows the execution time when using the distance matrix method, when computing the nearest centroid.

Chapter 6

Discussion

The MapReduce k-means clustering algorithm in this thesis found general player behaviour in a multiple batches of real game data. Results showed quality and stable clusters, the algorithm used efficient MR strategies by implementing a Combiner phase to reduce traffic and efficient vectorisation operations computing the nearest centroid in the Mapper phase. Scalable both horizontal and vertical using Amazon EMR, running multiple on-demand instances on a Hadoop cluster.

The algorithm clusters each data batch n-iterations resulting in centroids describing the a general player behaviour of each cluster, and incrementally clusters next incoming data batches by using the output from the previous batch as input to the next batch. So each data batch contributes to the general player behavior for the next batch.

Iterating only once per game data batch with the MR k-means algorithm resulted in higher quality and more stable clusters than increasing the number of iterations per batch. Allowing the centroids move slowly beetween different data batches, instead taking aggressive steps towards a data that will change in the next batch.

Three player profiles were interpreted from the last game data batch, using the centroids of the clusters to describe the general player behaviours, after incrementally clustering the previous data batches. These player profiles were not evaluated by a game developer that developed the game under research but instead it was evaluated using the authors intuition.

Evaluating the algorithm on a fictive dataset gave interesting results, where there was completely different behaviour, but was a fictive one. Three clusters are on a aggressive move and changes every data batch was a little bit more challence. The Iteration 2 version was ranked highest in cluster quality and error stability, but Iteration 1 was not far way.

The weakness for the algorithm is that it was not evaluated on larger multiple set of data batches, both in numbers and volume. Such that these results just indicate that for a real game data and a similar game it would be good to use one iteration and slowly after incrementally processing multiple batches the centroids will give high quality clusters and give a good representation of the general behaviors of the population. Also the k number of clusters to be found by k-means was not tested with different numbers than 3, this would be ofcourse be problem specific. Also interesting is to have the k to adjust to evolving behaviors in the data. Looking into iterative MapReduce frameworks where there is possible to run n iterations inside MapReduce instead of launching a MapReduce job each time (more overhead), would be a good research.

The MR k-means algorithm can be applied to any game data and scales out when increasing the dataset size.

Chapter 7

Conclusions

Data is all around us and is inevitable, it has been around for a quite some time but is getting bolder and more complex to understand and work with. Using services like Amazon Elastic MapReduce web service and their simple web storage (S3) is a right step analyzing and understanding these gigantic volumes of data. Implementing algorithms that can be distributed and executed in parallel is a very powerful tool when using it on large-scale data.

Free online games (Free-to-Play) are increasing each day through, e.g. Facebook and Google Plus, generating revenue through in-game transactions. Data analysis and better understanding the customers and their player behaviour is a vital information to keep the customer in the game and to buy more virtual items using micro-transactions with real money. The games can be very complex and offer wide range of events and behaviour where millions of users interact to each other.

The goals for this thesis were achieved and the contribution are the following:

- A MapReduce k-means algorithm that can find general player behaviour in the real game data set provided by GameAnalyltics, by incrementally cluster multiple batches. And were interpreted to player profiles.
- The algorithm finds stable and quality clusters while incrementally clustering multiple data batches, both for real game data and fictive generated datasets, where centroids move rapidly between batches.

- The algorithm uses an efficient MR implementation function called Combiner, to minimize network and data traffic inside the MapReduce framework.
- The algorithm uses efficient computation vectorisation approaches to perform fast operations on array like data objects, when computing the nearest centroid in the Mapper phase inside MR.

7.0.1 Future Work

- Evaluate the algorithm over larger time periods, more number of data batches and in larger size and for different real life datasets. Interesting would to see if the algorithm continues to be stable.
- Evaluate which number of iterations would give the best results to certain types of evolving datasets.
- Looking into Iterative MapReduce frameworks that are much more efficient to deal with many iterations, by running a internal loop, meaning a separate control program is redundant.
- Evaluate if there is a possibility to use a distance matrix computation efficiently to find the nearest centroid running on Amazon EMR, without running into memory troubles. E.g. using more powerful *Master Controller* instance, or higher memory instance running on Hadoop.

Bibliography

[1] Anders Drachen, Magy Seif El-Nasr, and Alessandro Canossa. Game analytics - the basics. In Magy Seif El-Nasr, Anders Drachen, and Alessandro Canossa, editors, *Game Analytics*, pages 13–40. Springer London, 2013. ISBN 978-1-4471-4768-8. doi: 10.1007/978-1-4471-4769-5˙2.

[2] A. Drachen, C. Thurau, R. Sifa, and C. Bauckhage. A comparison of methods for player clustering via behavioral telemetry. In *Foundations of Digital Games 2013*, 2013.

[3] Jun H. Kim, Daniel V. Gunn, Eric Schuh, Bruce Phillips, Randy J. Pagulayan, and Dennis Wixon. Tracking real-time user experience (true): a comprehensive instrumentation solution for complex systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 443–452, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357126.

[4] Anders Drachen and Alessandro Canossa. Evaluating motion: Spatial user behaviour in virtual environments. *International Journal of Arts and Technology*, 4 (3):294–314, 2011. doi: 10.1504/IJART.2011.041483.

[5] Randy J. Pagulayan, Kevin Keeker, Dennis Wixon, Ramon L. Romero, and Thomas Fuller. User-centered design in games. In Julie A. Jacko and Andrew Sears, editors, *The human-computer interaction handbook*, chapter User-centered design in games, pages 883–906. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2003. ISBN 0-8058-3838-4. URL `http://dl.acm.org/citation.cfm?id=772072.772128`.

[6] M Seif El-Nasr and Canossa A Drachen A. *Game Analytics: Maximizing the Value of Player Data*. Springer, 2013. ISBN 978-1-4471-4768-8.

[7] Geogios N. Yannakakis. Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 285–292, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212954.

[8] A. Drachen, R. Sifa, C. Bauckhage, and C. Thurau. Guns, swords and data: Clustering of player behavior in computer games in the wild. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 163–170, 2012. doi: 10.1109/CIG.2012.6374152.

[9] Tim Fields and Brandon Cotton. *Social Game Design: Monetization Methods and Mechanics*. CRC Press, 12 2011. ISBN 978-0240817668.

[10] Rui Xu and II Wunsch, D. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005. ISSN 1045-9227. doi: 10.1109/TNN.2005.845141.

[11] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.

[12] A. Drachen, A. Canossa, and G.N. Yannakakis. Player modeling using self-organization in tomb raider: Underworld. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 1–8, 2009. doi: 10.1109/CIG.2009.5286500.

[13] T. Mahlmann, A. Drachen, J. Togelius, A. Canossa, and G.N. Yannakakis. Predicting player behavior in tomb raider: Underworld. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 178–185, 2010. doi: 10.1109/ITW.2010.5593355.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

[15] C. Thurau and C. Bauckhage. Analyzing the evolution of social groups in world of warcraft ő. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 170–177, 2010. doi: 10.1109/ITW.2010.5593358.

[16] GameAnalytics Aps. Data and analytics engine for game studios, 2013. URL http://www.gameanalytics.com.

[17] G Zoeller. Game development telemetry. In *Proceedings of the Game Developers Conference*, 2010.

[18] Anders Drachen, Christian Thurau, Julian Togelius, GeorgiosN. Yannakakis, and Christian Bauckhage. Game data mining. In Magy Seif El-Nasr, Anders Drachen, and Alessandro Canossa, editors, *Game Analytics*, pages 205–253. Springer London, 2013. ISBN 978-1-4471-4768-8. doi: 10.1007/978-1-4471-4769-5˙12.

[19] Rui Xu and Don Wunsch. *Clustering*. Wiley-IEEE Press, 2009. ISBN 9780470276808.

[20] E. W. Forgy. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965. URL http://ci.nii.ac.jp/naid/10009668881/en/.

[21] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symp. Mathematical Statist. Probability*, pages 281–297, 1967. URL http://www-m9.ma.tum.de/foswiki/pub/WS2010/CombOptSem/kMeans.pdf.

[22] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

[23] Y. Linde, A. Buzo, and R.M. Gray. An algorithm for vector quantizer design. *Communications, IEEE Transactions on*, 28(1):84–95, 1980. ISSN 0090-6778. doi: 10.1109/TCOM.1980.1094577.

[24] Allen Gersho and Robert M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991. ISBN 0-7923-9181-0.

[25] J. Han, M. Kamber, and J. Pei. *Data Mining, Second Edition: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2006. ISBN 9780080475585. URL http://books.google.dk/books?id=AfL0t-YzOrEC.

[26] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009. ISSN 0885-6125. doi: 10.1007/s10994-009-5103-0.

[27] Anders Drachen and Alessandro Canossa. Towards gameplay analysis via gameplay metrics. In *Proceedings of the 13th International MindTrek Conference: Everyday Life in the Ubiquitous Era*, MindTrek '09, pages 202–209, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-633-5. doi: 10.1145/1621841.1621878.

[28] R.D. Schneider. *Hadoop for Dummies*. John Wiley & Sons Canada, Limited, 2012. ISBN 9781118250518. URL http://books.google.ca/books?id=0xhYLwEACAAJ.

[29] Andrzej Bialecki, Michael Cafarella, Doug Cutting, and Owen OŠMALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. 11, 2005. URL http://lucene.apache.org/hadoop.

[30] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011. ISBN 1107015359, 9781107015357.

[31] Tim Marsh, Shamus Smith, Kiyoung Yang, and Cyrus Shahabi. Continuous and unobtrusive capture of User-Player behaviour and experience to assess and inform game design and development. In *1st World Conference for Fun 'n Games*, Preston, England, 2006.

[32] Olana Missura and Thomas Gärtner. Player modeling for intelligent difficulty adjustment. In João Gama, VítorSantos Costa, AlípioMário Jorge, and PavelB. Brazdil, editors, *Discovery Science*, volume 5808 of *Lecture Notes in Computer Science*, pages 197–211. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04746-6. doi: 10.1007/978-3-642-04747-3˙17. URL http://dx.doi.org/10.1007/978-3-642-04747-3_17.

[33] C. Thurau, K. Kersting, and C. Bauckhage. Convex non-negative matrix factorization in the wild. In *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, pages 523–532, 2009. doi: 10.1109/ICDM.2009.55.

[34] K. Kersting, M. Wahabzada, C. Thurau, and C. Bauckhage. Hierarchical convex nmf for clustering massive data. In Qiang Yang Masashi Sugiyama, editor, *Proceedings of the 2nd Asian Conference on Machine Learning (ACML–10)*, Tokyo, Japan,

Nov 8–10 2010. URL http://www-kd.iai.uni-bonn.de/pubattachments/477/kersting10acml.pdf. draft.

[35] Anil K. Jain. Data clustering: 50 years beyond k-means. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ECML PKDD '08, pages 3–4, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87478-2. doi: 10.1007/978-3-540-87479-9˙3.

[36] Lior Rokach. A survey of clustering algorithms. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 269–298. Springer US, 2010. ISBN 978-0-387-09822-7. doi: 10.1007/978-0-387-09823-4˙14.

[37] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the eighteenth annual symposium on Computational geometry*, SCG '02, pages 10–18, New York, NY, USA, 2002. ACM. ISBN 1-58113-504-1. doi: 10.1145/513400.513402.

[38] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-24-5. URL http://dl.acm.org/citation.cfm?id=1283383.1283494.

[39] Nir Ailon, Ragesh Jaiswal, and Claire Monteleoni. Streaming k-means approximation. *Advances in Neural Information Processing Systems*, 22:10–18, 2009. URL http://www1.cs.columbia.edu/~rjaiswal/ajmNIPS09.pdf.

[40] Vladimir Braverman, Adam Meyerson, Rafail Ostrovsky, Alan Roytman, Michael Shindler, and Brian Tagiku. Streaming k-means on well-clusterable data. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 26–40. SIAM, 2011. URL http://dl.acm.org/citation.cfm?id=2133036.2133039.

[41] Michael Shindler, Alex Wong, and Adam W. Meyerson. Fast and accurate k-means for large datasets. In J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*

24, pages 2375–2383. NIPS, 2011. URL http://books.nips.cc/papers/files/nips24/NIPS2011_1271.pdf.

[42] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 674–679, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10664-4. URL http://dx.doi.org/10.1007/978-3-642-10665-1_71.

[43] Makho Ngazimbi. Data clustering using mapreduce. Master of science in computer science, Boise State University, March 2009. URL http://cs.boisestate.edu/~amit/research/makho_ngazimbi_project.pdf.

[44] Georgios Christopoulos. Fast, parallel stream clustering using hadoop online. Diploma thesis, Technical University of Crete, July 2011. URL http://titan.softnet.tuc.gr:8080/softnet/GetFile?FILE_TYPE=PUB.FILE&FILE_ID=201.

[45] Grace Nila Ramamoorthy. K-means clustering using hadoop mapreduce. Msc advanced software engineering in computer science, University College Dublin, September 2011. URL http://www.resumegrace.appspot.com/pdfs/kmeansCluster.pdf.

[46] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O"Callaghan. Clustering data streams: Theory and practice. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):515–528, 2003. ISSN 1041-4347. doi: 10.1109/TKDE.2003.1198387.

[47] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 378–388, 1999. doi: 10.1109/SFFCS.1999.814609.

[48] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *SIGMOD Rec.*, 25(2):103–114, June 1996. ISSN 0163-5808. doi: 10.1145/235968.233324.

[49] David Marchette. A statistical method for profiling network traffic. In *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring - Volume 1*, ID'99, pages 13–13, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267880.1267893.

[50] Marcel R. Ackermann, Christiane Lammersen, Marcus Märtens, Christoph Raupach, Christian Sohler, and Kamil Swierkot. Streamkm++: A clustering algorithms for data streams. In *ALENEX*, pages 173–187, 2010. URL http://www.siam.org/proceedings/alenex/2010/alx10_016_ackermannm.pdf.

[51] A. Meyerson. Online facility location. In *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, FOCS '01, pages 426–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1390-5. URL http://dl.acm.org/citation.cfm?id=874063.875567.

[52] Charu C. Aggarwal. An intuitive framework for understanding changes in evolving data streams. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 261–, 2002. doi: 10.1109/ICDE.2002.994715.

[53] Charu C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 575–586, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872826.

[54] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003. ISBN 0-12-722442-4. URL http://dl.acm.org/citation.cfm?id=1315451.1315460.

[55] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 852–863. VLDB Endowment, 2004. ISBN 0-12-088469-0. URL http://dl.acm.org/citation.cfm?id=1316689.1316763.

[56] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 61–72, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8. doi: 10.1145/304182.304188.

[57] Aoying Zhou, Feng Cao, Weining Qian, and Cheqing Jin. Tracking clusters in evolving data streams over sliding windows. *Knowl. Inf. Syst.*, 15(2):181–214, May 2008. ISSN 0219-1377. doi: 10.1007/s10115-007-0070-x.

[58] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002. doi: 10.1137/S0097539701398363.

[59] Hai-Guang Li, Gong-Qing Wu, Xue-Gang Hu, Jing Zhang, Lian Li, and Xindong Wu. K-means clustering with bagging and mapreduce. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–8, 2011. doi: 10.1109/HICSS.2011.265.

[60] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996. ISSN 0885-6125. doi: 10.1023/A:1018054314350.

[61] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855711.1855732.

[62] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851593.

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[64] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2): 285–296, September 2010. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1920841.1920881.

[65] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21 (2):169–190, April 2012. ISSN 1066-8888. URL http://dx.doi.org/10.1007/s00778-012-0269-7.

[66] Cairong Yan, Xin Yang, Ze Yu, Min Li, and Xiaolin Li. Incmr: Incremental data processing based on mapreduce. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 534–541, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4755-8. URL http://dx.doi.org/10.1109/CLOUD.2012.67.

[67] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. URL http://doi.acm.org/10.1145/2038916.2038923.

[68] Pramod Bhatotia, Marcel Dischinger, Rodrigo Rodrigues, and Umut A Acar. Slider: Incremental sliding-window computations for large-scale data analysis. *MPI-SWS-2012-004*, September 2012. URL http://www.mpi-sws.org/tr/2012-004.pdf.